GNU Serveez

A server framework
Edition 0.3.1, 11 December 2021

Stefan Jahn
Raimund Jacob
Thien-Thi Nguyen

This manual documents GNU Serveez 0.3.1, released 11 December 2021.

Table of Contents

1 Using Serveez

We know, you usually don't read the documentation. Who does. But please, read at the very least this chapter. It contains information on the basic concepts. Larger parts of the manual can be used as a reference manual for the various servers.

1.1 Building and installing

1.1.1 Rebuild the package from the sources

You can skip this section if you are familiar with the GNU'ish way of configuring, compiling and installing a GNU package.

1.1.1.1 Getting the source

Serveez can be found on `https://ftp.gnu.org/gnu/serveez/`, on one of the mirrors (`https://www.gnu.org/prep/ftp.html`) or at its original location `http://www.lkcc.org/~ela/download/`.

1.1.1.2 Requirements

Serveez needs GNU Guile (Ubiquitous Intelligent Language for Extensions). The current version of Serveez is known to work with Guile 1.3 and later. Guile can be downloaded at `https://ftp.gnu.org/gnu/guile/`.

When installing Guile, consider specifying to its `configure` script options along the lines of:

```
--enable-static --disable-shared
--disable-debug-freelist
--disable-debug-malloc
--disable-guile-debug
--disable-arrays --disable-posix
--enable-networking --disable-regex
--without-threads --enable-ltdl-convenience
```

This option set is tuned for Guile 1.4 and may or may not work for your particular installation; you may need to experiment a bit. Most important is that the program '`guile-config`' be findable in a directory named in the `PATH` environment variable. If not, configuration will fail with message "Guile not found".

1.1.1.3 Installation

Unpack the distribution tarball:

```
sleepless ~> gzip -cd serveez-0.3.1.tar.gz | tar xvf -
```

Change into the source directory:

```
sleepless ~> cd serveez-0.3.1
```

Configure the source package for your system:

Normally, this is done by running the `configure` script in the top-level directory. GNU Serveez needs a reasonably conformant C99 compiler to build. If the

`configure` script is not able to automatically find and enable such a compiler, you can specify it directly using the '`CC`' command-line option. For example:

```
sleepless ~> ./configure --prefix /gnu CC='/gnu/bin/gcc'
```

Note that in this example you can achieve the same results by making sure `/gnu/bin` is in the `PATH` env var.

In previous Serveez releases, the `configure` script had some builtin compiler flags for warnings and optimizations, conveniently exposed via command-line options. These are no longer available; instead, you can use `CFLAGS` for such purposes (see below). Here is a complete list of the `configure` script options. The list of known options can the obtained via '`./configure --help`'.

'`--enable-debug`'
: All of the debug messages (debug: some annoying crap text) can be suppressed by setting the debug level (-v). If you do not want these messages built in at all then disable this feature.

'`--enable-control-proto`'
: If you enable this feature the control protocol will be supported by Serveez. This protocol is for remote control of the server.

'`--enable-irc-proto`'
: Enabling this feature tells the software package to support the IRC (Internet Relay Chat) protocol.

'`--enable-irc-ts`'
: This feature is only available if you enabled the IRC protocol. If you enabled both of them then Serveez will support the so called TimeStamp protocol, which is an EFNet extension of the original IRC protocol. In order to connect to EFNet you MUST use this option.

'`--enable-http-proto`'
: When using Serveez as part of the textSure (C) chat system you will will have need of an additional web server. This option makes Serveez support a simple HTTP protocol.

'`--enable-flood`'
: If you enable this feature Serveez will support a simple built-in flood protection. It is always useful to protect the software from flood clients.

'`--with-mingw=DIR`'
: When compiling under M$-Windows the *DIR* argument specifies the path to the extra MinGW32 library and header files. If you want the final executable to use the Cygwin project's `cygwin1.dll` instead, you have to disable this option by passing the configure script '`--without-mingw`' or '`--with-mingw=no`'.

'`--enable-sntp-proto`'
: This option enables support for a simple network time protocol server.

‘`--enable-poll`’
: If the target system supports `poll` and this feature is enabled the main file descriptor loop is done via `poll`. This helps to work around the (g)libc's file descriptor limit. Otherwise Serveez always falls back to the `select` system call.

‘`--enable-sendfile`’
: This option enables the use of the `sendfile` system call. Disabling it using ‘`--disable-sendfile`’ provides a work-around for bogus implementations of `sendfile`.

‘`--enable-gnutella`’
: If you do *not* want the Gnutella spider client compiled in you need to *disable* this option.

‘`--enable-crypt`’
: This option tells Serveez to process any passwords as `crypt`ed.

‘`--enable-tunnel`’
: If you enable this feature the port forwarder will be included. This is useful if you plan to use Serveez as a gateway or firewall workaround.

‘`--enable-fakeident`’
: By enabling this you will get a fake ident server included in the binary executable.

‘`--enable-guile-server`’
: If you enable this feature the user is able to write servers using Guile.

‘`--enable-passthrough`’
: This includes the program passthrough server in the Serveez binary. The server provides basic inetd functionality.

‘`--enable-iflist`’
: If Serveez is unable to detect the correct list of local network interfaces (‘`serveez -i`’) you can disable this option and setup them manually in the configuration file.

‘`--enable-heap-count`’
: This option depends on ‘`--enable-debug`’. With the debugging option disabled there is also no support for heap counters. The heap counters are used to detect memory leaks in Serveez.

‘`--enable-libserveez-install`’
: This causes ‘`make install`’ to also copy libserveez and its header files to `$(libdir)` and `$(includedir)`, respectively. While Serveez is in alpha (version less than ‘`0.8.0`’), this option is disabled by default.

For maximum flexibility and reproducibility, we recommend enabling warnings and optimizations specifying `CFLAGS` as a command-line option to the

`configure` script. For reference, here is the collected set of flags built into previous Serveez releases:

(warnings)

```
-W -fullwarn -pedantic -ansi
-Wall -Wcast-align -Wstrict-prototypes
-Wformat -Wno-unused -Wno-long-long
```

(optimizations)

```
-O2 -fomit-frame-pointer -fstrength-reduce
-funroll-loops -finline-functions
-fexpensive-optimizations -fcaller-saves
-frerun-loop-opt -foptimize-register-move
-ffunction-cse -fpeephole -momit-leaf-frame-pointer
-fschedule-insns2 -m486 -march=pentiumpro -O3
```

Note that not all flags may be compatible with your compiler or even each other. That's one of the reasons we no longer take this approach. See Section "Defining Variables" in *The Autoconf Manual*.

```
sleepless ~/serveez-0.3.1> ./configure \
  CFLAGS='-g -O3 -Wall -Wextra'
```

Now compile the package:

```
sleepless ~/serveez-0.3.1> make
```

Install serveez:

You must have root privileges if you want to install the package in the standard location `/usr/local` or in any location that is only writable by root.

```
sleepless ~/serveez-0.3.1> make install
```

If you have problems building the package out of the box this is due to GNU libtool's inability to handle dynamic linking in most cases. That is why we recommend to try to configure the package with '`--disable-shared`'.

1.2 Environment variables

When using the serveezopt package or playing around with the dynamic server module loader of Serveez you can tell the core API of Serveez (which is the `libserveez.[so|dll]` library) to use an additional load path to find these server modules. The environment variable '`SERVEEZ_LOAD_PATH`' holds this information. You can set it up via:

```
  on Unices
sleepless ~> export SERVEEZ_LOAD_PATH=/home/lib:/usr/local/lib

  or on Windows
C:\HOME> set SERVEEZ_LOAD_PATH=C:\HOME\LIB;C:\USR\LOCAL\LIB
```

1.3 Starting Serveez

When Serveez is started it reads its configuration from a file called `serveez.cfg` in the current directory and runs the server loop afterwards. Press `^C` to abort the execution

of this program. Serveez is not interactive and does not automatically detach from the terminal.

1.4 Command line options

`-h, --help`
: Display this help and exit.

`-V, --version`
: Display version information and exit.

`-L, --list-servers`
: Display prefix and description of each builtin server, one per line, and exit. If Serveez was configured to include the Guile server (see Section 1.1 [Build and install], page 1), the output includes an additional line:

```
(dynamic)      (servers defined in scheme)
```

`-i, --iflist`
: List local network interfaces and exit.

`-f, --cfg-file=FILENAME`
: File to use as configuration file (serveez.cfg).

`-v, --verbose=LEVEL`
: Set level of logging verbosity.

`-l, --log-file=FILENAME`
: Use `FILENAME` for logging (default is stderr).

`-P, --password=STRING`
: Set the password for control connections. This option is available only if the control protocol is enabled. See Section 3.4.3 [Control Protocol Server], page 42.

`-m, --max-sockets=COUNT`
: Set the maximum number of socket descriptors.

`-d, --daemon`
: Start as daemon in background.

`-c, --stdin`
: Use standard input as configuration file.

`-s, --solitary`
: Do not start any builtin coserver instances.

1.5 Configuring Serveez

As noted above Serveez is configured via a configuration file which is by default `serveez.cfg` and can be set by passing the '`-f`' command line argument. When you pipe a file into Serveez or pass the '`-c`' argument on the command line the input stream will be used as configuration file no matter whether you passed a '`-f`' command line switch or not.

To make configuring more fun we did not invent yet another configuration file format. Instead we use a dialect of the Scheme programming language called GNU Guile (`https://`

www.gnu.org/software/guile/). There is no need to be worried if you are not a programmer. What you have to do is really simple and this document shows you everything you need to know. We also provide many examples. However there are some simple concepts you have to understand. The following paragraphs will explain them.

The idea of the configuration file is this: Serveez starts, runs the configuration file (other applications usually just read them and remember the settings) and finally enters its main loop doing the things you wanted it to.

There are three things you have to do in the configuration file.

1.5.1 Define ports

A `port` (in Serveez) is a transport endpoint. You might know them from other TCP or UDP server applications. For example: web servers (HTTP) usually listen on TCP port 80. However, there is more than TCP ports: we have UDP, ICMP and named pipes each with different options to set. Every port has a unique name you assign to it. The name of the port is later used to bind servers to it.

The following examples show how you setup different types of port configurations. You start to define such a port using the procedure `define-port!`. The first argument specifies the name of the port configuration. The remaining argument describes the port in detail.

1.5.1.1 Port configuration items

This table describes each configuration item for a port in Serveez. Note that not each item applies to every kind of port configuration.

`proto (string)`
: This is the main configuration item for a port configuration setting up the type of port. Valid values are '`tcp`', '`udp`', '`icmp`', '`raw`' and '`pipe`'. This configuration item decides which of the remaining configuration items apply and which do not.

`port (integer in the range 0..65535)`
: The `port` item determines the network port number on which TCP and UDP servers will listen. Thus it does not make sense for ICMP and named pipes. If you pass '`0`' Serveez will determine a free port in the range between 1 and 65535.

`recv (string or associative list)`
: This item describes the receiving (listening) end of a named pipe connection, i.e., the filename of a fifo node to which a client can connect by opening it for writing. Both the `recv` and `send` item apply to named pipes only. The value can either be an associative list or a simple filename. Using a simple filename leaves additional options to use default values. They deal mainly with file permissions and are described below.

`send (string or associative list)`
: This item is the sending end of a named pipe connection. It is used to send data when the receiving (listening) end has detected a connection. The following table enumerates the additional options you can setup if you pass an associative list and not a simple filename.

`name (string)`
: The filename of the named pipe. On Windows systems you can also specify the hostname on which the pipe should be created in the format '`\\hostname\pipe\name`'. By default (if you leave the leading '`\\hostname\pipe\`' part) the pipe will be created on '`\\.\pipe\name`' which refers to a pipe on the local machine.

`permission (octal integer)`
: This specifies the file permissions a named pipe should be created with. The given number is interpreted in a Unix'ish style (e.g., '`#o0666`' is a permission field for reading and writing for the creating user, all users in the same group and all other users).

`user (string)`
: The file owner (username) of the named pipe in textual form.

`group (string)`
: The file owner group (groupname) of the named pipe in textual form. If this item is left it defaults to the file owner's primary group.

`uid (integer)`
: The file owner of the named pipe as a user id. You are meant to specify either the `uid` item or the `user` item. Serveez will complain about conflicting values.

`gid (integer)`
: The file owner group of the named pipe as a group id. This item defaults to the file owner's primary group id. You are meant to specify either the `gid` item or the `group` item. Serveez will croak about conflicting values.

`ipaddr (string)`
: This configuration item specifies the IP address (either in dotted decimal form e.g., '`192.168.2.1`' or as a device description which can be obtained via '`serveez -i`') to which a server is bound to. The '`*`' keyword for all known IP addresses and the '`any`' keyword for any IP address are also valid values. The default value is '`*`'. The configuration item applies to network ports (TCP, UDP and ICMP) only.

`device (string)`
: The `device` configuration item also refers to the IP address a server can be bound to. It overrides the `ipaddr` item. Valid values are network device descriptions (probably no aliases and no loopback devices). It applies to network ports (TCP, UDP and ICMP) only.

  A note on device bindings: Device bindings are based on the `SO_BINDTODEVICE` socket layer option. This option is not available on all systems. We only tested it on GNU/Linux (2.2.18 and 2.4.17 as of this writing). Device bindings are very restrictive: only root can do it and only physical devices are possible. The loopback device cannot be used and no interface alias (i.e., '`eth0:0`'). A device binding can only be reached from the physical outside but it includes all aliases

for the device. So if you bind to device '`eth0`' even '`eth0:0`' (and all other aliases) are used. The connection has to be made from a remote machine. The advantage of this kind of binding is that it survives changes of IP addresses. This is tested for ethernet networks (i.e., eth*) and isdn dialups (i.e., ippp*). It does not work for modem dialups (i.e., ppp*) (at least for Stefan's PCMCIA modem). The problem seems to be the dialup logic actually destroying ppp*. Other opinions are welcome. Device bindings always win: If you bind to '`*`' (or an individual IP address) and to the corresponding device, connections are made with the device binding. The order of the `bind-server!` statements do not matter. This feature is not thoroughly tested.

`backlog (integer)`
: The `backlog` parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error. This parameter applies to TCP ports only.

`type (integer in the range 0..255)`
: This item applies to ICMP ports only. It defines the message type identifier used to send ICMP packets (e.g., '`8`' is an echo message i.e., PING).

`send-buffer-size (integer)`
: The `send-buffer-size` configuration item defines the maximum number of bytes the send queue of a client is allowed to grow to. The item influences the "send buffer overrun error condition". For packet oriented protocols (UDP and ICMP) you need to specify at least the maximum number of bytes a single packets can have. For UDP and ICMP this is 64 KByte. The value specified here is an initial value. It is used unless the server bound to this port changes it.

`recv-buffer-size (integer)`
: The `recv-buffer-size` configuration item defines the maximum number of bytes the receive queue of a client is allowed to grow to. The item influences the "receive buffer underrun error condition". The value specified here is an initial value. It is used unless the server bound to this port changes it.

`connect-frequency (integer)`
: This item determines the maximum number of connections per second the port will accept. It is a kind of "hammer protection". The item is evaluated for each remote client machine separately. It applies to TCP ports.

`allow (list of strings)`
: Both the `allow` and `deny` lists are lists of IP addresses in dotted decimal form (e.g., '`192.168.2.1`'). The `allow` list defines the remote machines which are allowed to connect to the port. It applies to TCP ports.

`deny (list of strings)`
: The `deny` list defines the remote machines which are not allowed to connect to the port. Each connection from one of these IP addresses will be refused and shut down immediately. It applies to TCP ports.

1.5.1.2 TCP port definition

Definition of a TCP port configuration with the name `foo-tcp-port`. The enhanced settings are all optional including the *ipaddr* property which defaults to '`*`'. The *ipaddr* item can contain any form of a dotted decimal internet address, a '`*`', '`any`' or an interface description which you can obtain by running '`serveez -i`'.

```
(define-port! 'foo-tcp-port '(
    ;; usual settings
    (proto  . tcp)              ;; protocol is tcp
    (port   . 42421)            ;; network port 42421
    (ipaddr . *)                ;; bind to all known interfaces
    (device . eth0)             ;; bind to network card

    ;; enhanced settings
    (backlog           . 5)     ;; enqueue max. 5 connections
    (connect-frequency . 1)     ;; allow 1 connect per second
    (send-buffer-size  . 1024)  ;; initial send buffer size in bytes
    (recv-buffer-size  . 1024)  ;; initial receive buffer size in bytes

    ;; allow connections from these ip addresses
    (allow             . (127.0.0.1 127.0.0.2))

    ;; refuse connections from this ip address
    (deny              . (192.168.2.7))
  ))
```

1.5.1.3 Pipe port definition

Definition of a pipe port configuration with the name `foo-pipe-port`. When bound to a server it creates the receiving end and listens on that. If some client accesses this named pipe the server opens the sending end which the client has to open for reading previously.

The only mandatory item is the file name of each pipe. If you want to specify a user creating the named pipe (file ownership) use either the *user* or the *uid* setting. Same goes for the items *group* and *gid*.

```
(define-port! 'foo-pipe-port `(
    (proto . pipe)                  ;; protocol is named pipe

    ;; specify the receiving endpoint
    (recv . ((name . ".foo-recv")    ;; name of the pipe
             (permissions . #o0666)  ;; create it with these permissions
             (user . "calvin")       ;; as user "calvin"
             (uid . 50)              ;; with the user id 50
             (group . "heros")       ;; which is in the group "heros"
             (gid . 100)))           ;; with the group id 100

    ;; specify the sending endpoint
    (send . ((name . ".foo-send")
             (permissions . #o0666)
```

```
            (user . "hobbes")
            (uid . 51)
            (group . "stuffed")
            (gid . 101)))
))
```

1.5.1.4 ICMP port definition

Define an ICMP port configuration which will accept connections from the network interface '`127.0.0.1`' only and communicates via the message type 8 as described in the Section 3.4.7 [Tunnel Server], page 46, chapter. The name of this port configuration is `foo-icmp-port`. When you are going to bind some server to this kind of port you have to ensure root (or Administrator under Windows) privileges.

```
(define-port! 'foo-icmp-port '((proto  . icmp)
                               (ipaddr . 127.0.0.1)
                               (type   . 8)))
```

1.5.1.5 UDP port definition

Simple definition of a UDP port configuration with the name `foo-udp-port`.

```
(define-port! 'foo-udp-port `((proto . udp)
                              (port  . 27952)))
```

1.5.2 Define servers

A `server` (in Serveez) is a snippet of code that implements some protocol. There are many servers built into Serveez but you can implement your own, too. For example we provide a webserver implementing the Hypertext Transfer Protocol (HTTP). Each server has a different set of options you can change. You can have many instances of every server, each with a different set of options. For example: You can create a webserver on TCP port 42420 publishing the Serveez documentation and also have another webserver on a different port publishing something else. Every server has a unique name you assign to it. The name of the server is later used to bind it to a port.

The following example instantiates a server with the short name "foo". Each server in Serveez has got a short name. See Section 3.4 [Existing servers], page 36, for the details. This example demonstrates everything which is possible in server configurations. You start a definition of a server with the procedure `define-server!`. The following argument specifies the name of the server instance (in this case `foo-server`) which starts with the short name. The second argument describes the server in detail. Each configuration item is setup with a `(key . value)` pair where *key* is the name of the configuration item and *value* is the value which depends on the type of the item. See Section 3.3 [Configuring servers], page 35, for a detailed description of each type of value.

```
(define-server! 'foo-server '(
    (bar . 100)                          ;; number
    (reply . "Booo")                     ;; character string
    (messages .                          ;; list of strings
      ("Welcome to the foo test server."
       "This one echos your lines."))
    (ports . (5 6 7 8 9))                ;; list of numbers
```

```
  (port . foo-tcp-port)                    ;; a port configuration
  (assoc . (( "GNU" . "great" )            ;; associative list
            ( "Tree" . "tall" )))
  (truth . #f)                             ;; boolean value
))
```

Serveez provides a number of server types. Each of them has a short name. The name of the server instance has to begin with this short name followed by a dash (-). You can append any suffix then. In the example above "foo" is the short name and `foo-server` the name of the server instance.

1.5.3 Bind servers to ports

Finally you can bind servers to ports. When you do so the server you created listens on the port, accepts connections and serves clients. It does so as soon as Serveez enters its main loop right after running the configuration file. Serveez won't stop until you interrupt it (e.g., by pressing `^C` in the terminal you started it in).

This example binds the server `foo-server` (s.a.) to the port `foo-tcp-port` which was described above. Therefore you need to call the procedure `bind-server!` which takes two arguments specifying the name of a port configuration and a server instance. Both need to be defined before you can write this statement.

```
(bind-server! 'foo-tcp-port 'foo-server)
```

One of the main features of Serveez is that you can bind multiple servers to the same port. This for example is useful to pass braindead firewall configurations or proxy servers. It is also possible to bind servers to ports they are actually not designed for. This might be used for debugging servers or other funny things (again, think about the firewall). This is the point we have to warn you: Some protocols cannot share the same port (e.g., the tunnel server) and some protocols simply won't work on 'wrong' ports. Additionally, you will not get error messages when that happens. The server just will not work then.

1.5.4 Additional configuration possibilities

The three procedures `define-port!`, `define-server!` and `bind-server!` return `#t` on success and `#f` on failure. For your convenience we provide some more built-in procedures, some of which are based upon those above.

1.5.4.1 output

`fs` *s* [*args...*] [Scheme Procedure]

Return a string made by applying `simple-format #f` to *s* and *args*. For example:

```
(fs "~A-~S" 'foo 42)
⇒ "foo-42"
```

`println` [*object...*] [Scheme Procedure]

Do `display` on each *object*. Then, output a newline.

`printsln` *spacer* [*object...*] [Scheme Procedure]

For each *object*, do `display` on it and on *spacer*, as well. Then, output a newline.

1.5.4.2 augmenting

`interface-add!` *interface* [Scheme Procedure]
Add *interface* to the list of known network interfaces. You can get the list of known interfaces by running the shell command '`serveez -i`'. The *interface* argument must be in dotted decimal form (e.g., '`127.0.0.1`'). Serveez provides this procedure for systems where it is unable to detect the list of network interface automatically.

`loadpath-add!` [*dir...*] [Scheme Procedure]
Append *dir...* to the server modules load path.

`serveez-load` *filename* [Scheme Procedure]
Try to load *filename* (via `primitive-load`). If *filename* is not absolute, search for it in the list of directories returned by `serveez-loadpath`. Return `#t` if successful, `#f` otherwise.

1.5.4.3 abstractions

`bind-servers!` [*args...*] [Scheme Procedure]
Bind all servers and ports in *args* to each other. This is a cross-product operation; given *s* servers, and *p* ports, *s* * *p* bindings will be created.

`create-tcp-port!` *basename port* [Scheme Procedure]
Define a new TCP port named by concatenating *basename* and *port*. Return the new name.

`bind-tcp-port-range!` *from to* [*servers...*] [Scheme Procedure]
Bind the list of *servers* to simple TCP port configurations whose network ports range between *from* and *to* both inclusive.

`create-udp-port!` *basename port* [Scheme Procedure]
Define a new UDP port named by concatenating *basename* and *port*. Return the new name.

`bind-udp-port-range!` *from to* [*servers...*] [Scheme Procedure]
Bind the list of *servers* to simple UDP port configurations whose network ports range between *from* and *to* both inclusive.

1.5.4.4 rpc

`getrpcent` [Scheme Procedure]
Return the next RPC entry as a vector of the form: `#(`*name aliases program-number*`)`. *name* is a symbol, *aliases* is a list (possibly empty) of symbols, and *program-number* is an integer. If the list is exhausted, return `#f`.

`getrpcbyname` *name* [Scheme Procedure]
Return the RPC entry for *name*, a string. (FIXME: Should be able to handle a symbol, too.) If no such service exists, signal error.

`getrpcbynumber` *number* [Scheme Procedure]
Return the RPC entry for *number*, an integer. If no such service exists, signal error.

`setrpcent` [*stayopen*] [Scheme Procedure]
: Open and rewind the file `/etc/rpc`. If optional arg *stayopen* (an integer) is non-zero, the database will not be closed after each call to `getrpc` (or its derivatives `getrpcent`, `getrpcbyname`, `getrpcbynumber`).

`endrpcent` [Scheme Procedure]
: Close the file `/etc/rpc`.

1.5.4.5 misc

`serveez-verbosity` [*level*] [Scheme Procedure]
: Return the verbosity level (an integer). Optional arg *level* means set it to that level, instead. This setting is overridden by the command-line '`-v`' option.

`serveez-maxsockets` [*max*] [Scheme Procedure]
: Return the maximum number of open sockets permitted (an integer). Optional arg *max* means set it to that number, instead. This setting is overridden by the command-line '`-m`' option.

`serveez-passwd` [*pw*] [Scheme Procedure]
: Return the control password (a string). Optional arg *pw* sets it to that, instead. This effectively does nothing if the control protocol is not enabled.

We now have a closer look at the internals of Serveez. If you are not interested in that have a look at the existing servers (See Section 3.4 [Existing servers], page 36.).

2 Concept

2.1 Overall concept

The primary functionality Serveez provides is a framework for Internet services. It can act as a kind of server collection and may even replace super-servers such as the inetd daemon.

Its key features and benefits are:

- support for packet and connection oriented protocols
  Serveez currently supports two server types. TCP and named pipe servers are connection oriented servers. This type of server accepts a client's connection request and communicates with it using a dedicated connection. The format of the incoming and outgoing data streams are irrelevant to Serveez. Packet oriented servers (like UDP and ICMP) receive data packets and respond to the sender with data packets. A server in Serveez can detect whether it is bound to a packet or connection oriented port configuration and thus can act as the expected type.
- server and client functionality
  Besides a wide variety of server capabilities, Serveez also contains some client functionality. This may be necessary when a server is meant to establish a connection to another server in response to an incoming request. For example, imagine a protocol where a client tells the server, "Let me be the server. Please connect to this *host* at this *port*". You are also able to implement pure clients.
- platform portability
  When writing this software the authors were always mindful of platform portability. Serveez compiles and runs on various Unix platforms as well as on Windows systems. See Chapter 6 [Porting issues], page 83, for details. Most of the routines in the core library can be used without regard to the platform on which you are programming or the details of its underlying system calls. Exceptions are noted in the documentation. Platform portability also means that the server code you write will run on other systems, too.
- powerful configuration capabilities
  Server configuration has always been a complicated but very important issue. When Serveez starts up it runs the configuration file using the programming language Guile. In contradiction, other (server) applications just read their configuration files and remember the settings in it. This makes them powerful enough to adapt the Serveez settings dynamically. Using the Guile interpreter also means that you can split your configuration into separate files and load these, perhaps conditionally, from the main configuration file.
- easy server implementation
  Serveez is a server framework. When implementing a new server the programmer need pay little or no attention to the networking code but is free to direct his attention to the protocol the server is meant to support. That protocol may be an established one such as HTTP, or may be a custom protocol fitting the specific application's requirements. The Section 3.2 [Writing servers], page 18, section describes this process in detail.
- code reusability
  The Serveez package comes along with a core library (depending on the system this

is a static library, shared library or a DLL) and its API which contains most of the functionality necessary to write an Internet server. Most probably, a programmer can also use the library for other (network programming related) purposes.

- server instantiation and network port sharing
  Once you have written a protocol server and integrated into Serveez's concept of servers the user can instantiate (multiply) the server. At the first glimpse this sounds silly, but with different server configurations it does not. If, for example, an administrator wishes to run multiple HTTP servers with different document roots, Serveez will handle them all in a single process. Also if the same administrator wants to run a HTTP server and some other server on the same network port this is possible with Serveez. You can run a single server on different network ports, too.

2.2 I/O Strategy

Serveez's I/O strategy is the traditional `select` method. It is serving many clients in a single server thread. This is done by setting all network handles to non-blocking mode. We then use `select` to tell which network handles have data waiting. This is the traditional Unix style multiplexing.

An important bottleneck in this method is that a `read` or `sendfile` from disk blocks if the data is not in core at the moment. Setting non-blocking mode on a disk file handle has no effect. The same thing applies to memory-mapped disk files. The first time a server needs disk I/O, its process blocks, all clients have to wait, and raw non-threaded performance may go to waste.

Unfortunately, `select` is limited to the number of `FD_SETSIZE` handles. This limit is compiled into the standard library, user programs and sometimes the kernel. Nevertheless, Serveez is able to serve about one thousand and more clients on GNU/Linux, a hundred on Win95 and more on later Windows systems.

We chose this method anyway because it seems to be the most portable.

An alternative method to multiplex client network connections is `poll`. It is automatically used when '`configure`' finds `poll` to be available. This will work around the builtin (g)libc's `select` file descriptor limit.

2.2.1 Limits on open filehandles

`Any Unix` The limits set by `ulimit` or `setrlimit`.

`Solaris` See the Solaris FAQ, question 3.45.

`FreeBSD` Use sysctl -w kern.maxfiles=nnnn to raise limit.

`GNU/Linux`
See Bodo Bauer's /proc documentation. On current 2.2.x kernels,

```
echo 32768 > /proc/sys/fs/file-max
```

increases the system limit on open files, and

```
ulimit -n 32768
```

increases the current process' limit. We verified that a process on Linux kernel 2.2.5 (plus patches) can open at least 31000 file descriptors this way. It has also

been verified that a process on 2.2.12 can open at least 90000 file descriptors this way (with appropriate limits). The upper bound seems to be available memory.

`Windows 9x/ME`
: On Win9x machines, there is quite a low limit imposed by the kernel: 100 connections system wide (!). You can increase this limit by editing the registry key HKLM\System\CurrentControlSet\Services\VxD\MSTCP\MaxConnections. On Windows 95, the key is a DWORD; on Windows 98, it's a string. We have seen some reports of instability when this value is increased to more than a few times its default value.

`Windows NT/2000`
: More than 2000 connections tested. It seems like the limit is due to available physical memory.

2.3 Alternatives to Serveez's I/O strategy

One of the problems with the traditional `select` method with non-blocking file descriptors occurs when passing a large number of descriptors to the `select` system call. The server loop then goes through all the descriptors, decides which has pending data, then reads and handles this data. For a large number of connections (say, 90000) this results in temporary CPU load peaks even if there is no network traffic.

Along with this behaviour comes the problem of "starving" connections. Connections which reside at the beginning of the `select` set are processed immediately while those at the end are processed significantly later and may possibly die because of buffer overruns. This is the reason why Serveez features priority connection: it serves listening sockets first and rolls the order of the remaining connections. In this way, non-priority connections are handled in a "round robin" fashion.

Other server implementations solve these problems differently. Some start a new process for each connection (fully-threaded server) or split the `select` set into pieces and let different processes handle them (threaded server). This method shifts the priority scheduling to the underlying operating system. Another method is the use of asynchronous I/O based upon signals where the server process receives a signal when data arrives for a connection. The signal handler queues these events in order of arrival and the main server loop continuously processes this queue.

3 Server

3.1 Introduction to servers

Serveez is a kind of server server. It allows different protocol servers to listen on various TCP/UDP ports, on ICMP sockets and on named pipes. Servers are instantiated with a certain configuration. It is possible to run multiple different servers on the same port.

This chapter covers all questions about how to write your own Internet protocol server with this package. Most of the common tasks of such a server have got a generic solution (default routines) which could be "overridden" by your own routines. There are some examples within this package. They are showing the possibilities with this package and how to implement servers.

The '`foo`' server does not do anything at all and is of no actual use but could be a basis for a new protocol server. We are now going to describe how this specific server works. Eventually the reader might get an impression of what is going on.

For better understanding the text below we will use the following terminology:

server definition
: A server definition is a `svz_servertype_t` structure which contains server specific members like its name, different callbacks, a single default configuration and a list of configuration items which determines what can be configured.

server configuration
: A server configuration can be any kind of structure. The default server configuration must be specified within the server definition (see above). When instantiating a server (which is done via the configuration file) the configuration items specified in the server definition get processed and are put into a copy of the default configuration. Thus we get an instance.

server instance
: A server instance is a copy of the server definition including the modified server configuration. A server gets instantiated by the configuration file parser. The concept of server instances has been introduced because we wanted Serveez to have the following features. A single server can have multiple instances with different behaviour due to different server configurations. A server instance can be bound to multiple port configurations. Different server instances (of the same and/or different server type) can share the same port configuration.

port configuration
: A port configuration in Serveez is represented by the `svz_portcfg_t` structure. Depending on a shared flag it contains different type of information specifying a transport endpoint. See Section 1.5.1 [Define ports], page 6, for more on this topic. It also can be a special configuration item within a server configuration. This is necessary if a server needs for some reason a remote transport endpoint. A server instance does not usually need to know about the port configuration it is bound to.

3.2 Writing servers

Serveez is meant to be a server framework. That is why it supports various ways to implement Internet servers. First of all there are some servers already included in the main serveez package (see Section 3.4 [Existing servers], page 36). These are called '`Builtin servers`'. Another possibility to add a new server are '`Embedded servers`' which are shared libraries (or DLL's) containing the server implementation. These can be dynamically loaded by Serveez at runtime. This kind of server is also called '`Server modules`'. A third possibility are the '`Guile servers`' which allow even unexperienced schemers to write an Internet server.

This section describes each of the above possibilities in detail.

3.2.1 Embedded servers

The core library of Serveez can be used to write standalone server modules. Serveez defines a certain interface for shared libraries which contain such server modules. If Serveez detects an unknown server type (server type which is not builtin) in its configuration file it tries to load a shared library containing this server type during startup.

3.2.1.1 Prerequisites

In order to implement a server module you need an existing installation of Serveez. This can be achieved issuing the following commands:

```
$ ./configure --enable-shared --prefix=/usr/local
$ make
$ make install
```

After successful installation you are able to compile and link against the Serveez core API. The headers should be available in `/usr/local/include` and the library itself (`libserveez.so` or `libserveez.dll`) is located in `/usr/local/lib` if you passed the configure script '`--prefix=/usr/local`'.

3.2.1.2 Server definition

The interface mentioned in the introduction is defined via the extern declaration of the server type in the shared library of the server module. Imagine you want to implement a server type called '`foo`'. This requires the external symbol `foo_server_definition` in the shared library. You can achieve this inserting the following lines into your header file:

```
/* Either Unices.  */
extern svz_servertype_t foo_server_definition;

/* Or Windows.  */
__declspec (dllexport) extern svz_servertype_t foo_server_definition;
```

The data symbol `foo_server_definition` must be statically filled with the proper content (See Section 3.2.3 [Builtin servers], page 31.)

3.2.2 Guile servers

This section describes the Guile interface to Serveez which provides the ability to write servers with Guile. Of course, you could do this without any help from Serveez, but it

makes the task a lot easier. This interface reduces the Guile implementation of an Internet server to a simple data processor.

3.2.2.1 Underlying libserveez

`libserveez-features` [Scheme Procedure]
Return a list of symbols representing the features of the underlying libserveez. For details, See Section 5.2.1 [Library features], page 56.

3.2.2.2 Special Data Types

Serveez extends Guile by various new data types which represent internal data structures of Serveez's core API.

- `#<svz-servertype>` represents a server type.
- `#<svz-server>` represents a server (an instance of a server type).
- `#<svz-socket>` represents a socket structure.

3.2.2.3 Passing Binary Data

The new binary data type (`#<svz-binary>`) provides access to any kind of unstructured data. It manages the data exchange between Guile and Serveez. There are some conversion procedures for strings and lists which help to process this binary data in a more complex (guile'ish) way.

`binary->string` *binary* [Scheme Procedure]
Convert the given binary smob *binary* into a string. Return the string itself.

`string->binary` *string* [Scheme Procedure]
Convert the given *string* into a binary smob. The data pointer of the binary smob is marked as garbage which must be `free`'d in the sweep phase of the garbage collector.

`binary?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an instance of the binary smob type.

`list->binary` *list* [Scheme Procedure]
Convert the scheme list *list* into a binary smob. Each of the elements of *list* is checked for validity. The elements can be either exact numbers in a byte's range or characters.

`binary->list` *binary* [Scheme Procedure]
Convert the given binary smob *binary* into a scheme list. The list is empty if the size of *binary* is zero.

`binary-search` *binary needle* [Scheme Procedure]
Search through the binary smob *binary* for *needle*, which can be an exact number, character, string or another binary smob. Return `#f` if the needle could not be found, or a positive number indicating the position of the first occurrence of *needle* in the binary smob *binary*.

`binary-set!` *binary index value* [Scheme Procedure]
Set the byte at position *index* of the binary smob *binary* to the value given in *value* which can be either a character or an exact number.

`binary-ref` *binary index* [Scheme Procedure]
Obtain the byte at position *index* of the binary smob *binary*.

`binary-length` *binary* [Scheme Procedure]
Return the size in bytes of the binary smob *binary*.

`binary-concat!` *binary append* [Scheme Procedure]
Append either the binary smob or string *append* onto the binary smob *binary*. If *binary* has been a simple data pointer reference it is then a standalone binary smob as returned by `string->binary`.

`binary-subset` *binary start* [*end*] [Scheme Procedure]
Create a subset binary smob from the given binary smob *binary*. The range of this subset is specified by *start* and *end* both inclusive (thus the resulting size is *`end - start + 1`*). With a single exception: If *end* is not given or specified with -1, return all data until the end of *binary*.

`binary-reverse` *binary* [Scheme Procedure]
Return a new binary smob with the reverse byte order of the given binary smob *binary*.

`binary-reverse!` *binary* [Scheme Procedure]
Perform an in-place reversal of the given binary smob *binary* and return it.

`binary-long-ref` *binary index* [Scheme Procedure]
Return the `long` value of the binary smob *binary* at the array index *index*.

`binary-long-set!` *binary index value* [Scheme Procedure]
Set the `long` value of the binary smob *binary* at the array index *index* to the given value *value*. Return the previous (overridden) value.

`binary-int-ref` *binary index* [Scheme Procedure]
Return the `int` value of the binary smob *binary* at the array index *index*.

`binary-int-set!` *binary index value* [Scheme Procedure]
Set the `int` value of the binary smob *binary* at the array index *index* to the given value *value*. Return the previous (overridden) value.

`binary-short-ref` *binary index* [Scheme Procedure]
Return the `short` value of the binary smob *binary* at the array index *index*.

`binary-short-set!` *binary index value* [Scheme Procedure]
Set the `short` value of the binary smob *binary* at the array index *index* to the given value *value*. Return the previous (overridden) value.

`binary-char-ref` *binary index* [Scheme Procedure]
Return the `char` value of the binary smob *binary* at the array index *index*.

`binary-char-set!` *binary index value* [Scheme Procedure]
Set the `char` value of the binary smob *binary* at the array index *index* to the given value *value*. Return the previous (overridden) value.

3.2.2.4 Server Definition

In order to set up a new server type, you use the procedure `define-servertype!`. This procedure takes one argument which must be an associative list specifying the server type in detail. There are optional and mandatory elements you can set up in this alist.

The following example shows the overall syntax of this procedure:

```
(define-servertype! '(

  ;; Mandatory: server type prefix for later use in (define-server!)
  (prefix          . "foo")

  ;; Mandatory: server type description
  (description     . "guile foo server")

  ;; Mandatory for TCP and PIPE servers: protocol detection
  (detect-proto    . foo-detect-proto)

  ;; Optional: global server type initialisation
  (global-init     . foo-global-init)

  ;; Optional: server instance initialisation
  (init            . foo-init)

  ;; Optional: server instance finalisation
  (finalize        . foo-finalize)

  ;; Optional: global server type finalisation
  (global-finalize . foo-global-finalize)

  ;; Mandatory for TCP and PIPE servers: socket connection
  (connect-socket  . foo-connect-socket)

  ;; Optional: server instance info
  (info-server     . foo-info-server)

  ;; Optional: client info
  (info-client     . foo-info-client)

  ;; Optional: server instance reset callback
  (reset           . foo-reset)

  ;; Optional: server instance notifier
  (notify          . foo-notify)

  ;; Mandatory for UDP and ICMP servers: packet handler
  (handle-request  . foo-handle-request)
```

```
;; Mandatory: server type configuration (may be an empty list)
(configuration   . (

  ;; The server configuration is an alist (associative list) again.
  ;; Each item consists of an item name and a list describing the
  ;; item itself.
  ;; Syntax: (key . (type defaultable default))
  (foo-integer       . (integer  #t 0))
  (foo-integer-array . (intarray #t (1 2 3 4 5)))
  (foo-string        . (string   #t "default-foo-string"))
  (foo-string-array  . (strarray #t ("guile" "foo" "server")))
  (foo-hash          . (hash     #t (("foo" . "bar"))))
  (foo-port          . (portcfg  #t foo-port))
  (foo-boolean       . (boolean  #t #t))
))))
```

`define-servertype!` *args* [Scheme Procedure]

Define a new server type based on *args*. (If everything works fine you have a freshly registered server type afterwards.) Return `#t` on success.

3.2.2.5 Predefined Procedures

The following subset of procedures may be used to implement a Guile server. They should be used within the callbacks defined in the `define-servertype!` procedure. Each of these callbacks gets passed the appropriate arguments needed to stuff into the following procedures. Please have a look at the example Guile servers for the details.

`svz:sock?` *sock* [Scheme Procedure]

Return `#t` if the given cell *sock* is an instance of a valid `#<svz-socket>`, otherwise `#f`.

`svz:sock:check-request` *sock* [*proc*] [Scheme Procedure]

Set the `check-request` member of the socket structure *sock* to *proc*. Return the previously handler if there is any.

`svz:sock:check-oob-request` *sock* [*proc*] [Scheme Procedure]

Set the `check-oob-request` callback of the given socket structure *sock* to *proc*, returning the previous callback (if there was any set before). The callback is run whenever urgent data (out-of-band) has been detected on the socket.

`svz:sock:send-oob` *sock oob* [Scheme Procedure]

Send byte *oob* as urgent (out-of-band) data through the underlying TCP stream of TCP *sock*. Return `#t` on successful completion and otherwise (either it failed to send the byte or the passed socket is not a TCP socket) `#f`.

`svz:sock:handle-request` *sock* [*proc*] [Scheme Procedure]

Set the `handle-request` member of the socket structure *sock* to *proc*. Return the previously set handler if there is any.

`svz:sock:boundary` *sock boundary* [Scheme Procedure]
Setup the packet boundary of the socket *sock*. The given string value *boundary* can contain any kind of data. If *boundary* is an exact number, set up the socket to parse fixed sized packets. More precisely, set the `check-request` callback of the given socket structure *sock* to an internal routine which runs the socket's `handle-request` callback when it detects a complete packet specified by *boundary*.

For instance, you can arrange for Serveez to pass the `handle-request` procedure lines of text by calling `(svz:sock:boundary sock "\n")`.

`svz:sock:floodprotect` *sock* [*flag*] [Scheme Procedure]
Set or unset the flood protection bit of the given socket *sock*. Return the previous value of this bit (`#t` or `#f`). The *flag* argument must be either boolean or an exact number and is optional.

`svz:sock:print` *sock buffer* [Scheme Procedure]
Write *buffer* (string or binary smob) to the socket *sock*. Return `#t` on success and `#f` on failure.

`svz:sock:final-print` *sock* [Scheme Procedure]
Schedule the socket *sock* for shutdown after all data within the send buffer queue has been sent. You should call this right **before** the last call to `svz:sock:print`.

`svz:sock:no-delay` *sock* [*enable*] [Scheme Procedure]
Turn the Nagle algorithm for the TCP socket *sock* on or off depending on the optional *enable* argument. Return the previous state of this flag (`#f` if Nagle is active, `#t` otherwise). By default this flag is switched off. This socket option is useful when dealing with small packet transfer in order to disable unnecessary delays.

`svz:sock:send-buffer` *sock* [Scheme Procedure]
Return the send buffer of the socket *sock* as a binary smob.

`svz:sock:send-buffer-size` *sock* [*size*] [Scheme Procedure]
Return the current send buffer size and fill status in bytes of the socket *sock* as a pair of exact numbers. If the optional argument *size* is given, set the send buffer to the specified size in bytes.

`svz:sock:receive-buffer` *sock* [Scheme Procedure]
Return the receive buffer of the socket *sock* as a binary smob.

`svz:sock:receive-buffer-size` *sock* [*size*] [Scheme Procedure]
Return the current receive buffers size and fill status in bytes of the socket *sock* as a pair of exact numbers. If the optional argument *size* is given, set the receive buffer to the specified size in bytes.

`svz:sock:receive-buffer-reduce` *sock* [*length*] [Scheme Procedure]
Dequeue *length* bytes from the receive buffer of the socket *sock*, or all bytes if *length* is omitted. Return the number of bytes actually shuffled away.

`svz:sock:connect` *host proto* [*port*] [Scheme Procedure]
Establish a network connection to the given *host* [:*port*]. If *proto* equals `PROTO_ICMP` the *port* argument is ignored. Valid identifiers for *proto* are `PROTO_TCP`, `PROTO_UDP` and `PROTO_ICMP`. The *host* argument must be either a string in dotted decimal form, a valid hostname or an exact number in host byte order. When giving a hostname this operation might block. The *port* argument must be an exact number in the range from 0 to 65535, also in host byte order. Return a valid `#<svz-socket>` or `#f` on failure.

`svz:sock:disconnected` *sock* [*proc*] [Scheme Procedure]
Set the `disconnected-socket` member of the socket structure *sock* to *proc*. The given callback runs whenever the socket is lost for some external reason. Return the previously set handler if there is one.

`svz:sock:kicked` *sock* [*proc*] [Scheme Procedure]
Set the `kicked-socket` callback of the given socket structure *sock* to *proc* and return any previously set procedure. This callback gets called whenever the socket gets closed by Serveez intentionally.

`svz:sock:trigger` *sock* [*proc*] [Scheme Procedure]
Set the `trigger` callback of the socket structure *sock* to *proc* and return any previously set procedure. The callback is run when the `trigger-condition` callback returns `#t`.

`svz:sock:trigger-condition` *sock* [*proc*] [Scheme Procedure]
Set the `trigger-condition` callback for the socket structure *sock* to *proc*. Return the previously set procedure if available. The callback is run once every server loop indicating whether the `trigger` callback should be run or not.

`svz:sock:idle` *sock* [*proc*] [Scheme Procedure]
Set the `idle` callback of the socket structure *sock* to *proc*. Return any previously set procedure. The callback is run by the periodic task scheduler when the `idle-counter` of the socket structure drops to zero. If this counter is not zero it gets decremented once a second. The `idle` callback can reset `idle-counter` to some value and thus can re-schedule itself for a later task.

`svz:sock:idle-counter` *sock* [*counter*] [Scheme Procedure]
Return the socket structure *sock*'s current `idle-counter` value. If the optional argument *counter* is given, the set the `idle-counter`. Please have a look at the `svz:sock:idle` procedure for the exact meaning of this value.

`svz:sock:parent` *sock* [*parent*] [Scheme Procedure]
Return the given socket's *sock* parent and optionally set it to the socket *parent*. Return either a valid `#<svz-socket>` object or an empty list.

`svz:sock:referrer` *sock* [*referrer*] [Scheme Procedure]
Return the given socket's *sock* referrer and optionally set it to the socket *referrer*. Return either a valid `#<svz-socket>` or an empty list.

`svz:sock:server` *sock* [*server*] [Scheme Procedure]
Return the `#<svz-server>` object associated with the given argument *sock*. The optional argument *server* can be used to redefine this association and must be a valid `#<svz-server>` object. For a usual socket callback like `connect-socket` or `handle-request`, the association is already in place. But for sockets created by `svz:sock:connect`, you can use it in order to make the returned socket object part of a server.

`svz:sock:local-address` *sock* [*address*] [Scheme Procedure]
Return the current local address as a pair like `(host . port)` with both entries in network byte order. If you pass the optional argument *address*, you can set the local address of the socket *sock*.

`svz:sock:remote-address` *sock* [*address*] [Scheme Procedure]
Return the current remote address as a pair like `(host . port)` with both entries in network byte order. If you pass the optional argument *address*, you can set the remote address of the socket *sock*.

`svz:sock:find` *ident* [Scheme Procedure]
Return the `#<svz-socket>` specified by *ident*, a pair of integers in the form `(identification . version)`. If that socket no longer exists, return `#f`.

`svz:sock:ident` *sock* [Scheme Procedure]
Return a pair of numbers identifying the given `#<svz-socket>` *sock*, which can be passed to `svz:sock:find`. This may be necessary when you are passing a `#<svz-socket>` through coserver callback arguments in order to verify that the passed `#<svz-socket>` is still valid when the coserver callback runs.

`svz:sock:protocol` *sock* [Scheme Procedure]
Return one of the `PROTO_TCP`, `PROTO_UDP`, `PROTO_ICMP`, `PROTO_RAW` or `PROTO_PIPE` constants indicating the type of the socket structure *sock*. If there is no protocol information available, return `#f`.

`svz:read-file` *port size* [Scheme Procedure]
Return either a binary smob containing a data block read from the open input port *port* with a maximum number of *size* bytes, or the end-of-file object if the underlying ports end has been reached. The size of the returned binary smob may be less than the requested size *size* if it exceed the current size of the given port *port*. Throw an exception if an error occurred while reading from the port.

`svz:server?` *server* [Scheme Procedure]
Return `#t` if the given cell *server* is an instance of a valid `#<svz-server>`, otherwise `#f`.

`svz:server:listeners` *server* [Scheme Procedure]
Return a list of listening `#<svz-socket>` smobs to which the given server instance *server* is currently bound, or an empty list if there is no such binding yet.

`svz:server:clients` *server* [Scheme Procedure]
Return a list of `#<svz-socket>` client smobs associated with the given server instance *server* in arbitrary order, or an empty list if there is no such client.

`svz:server:config-ref` *server key* [Scheme Procedure]
Return the configuration item specified by *key* of the given server instance *server*. You can pass this procedure a socket, too, in which case the appropriate server instance is looked up. If the given string *key* is invalid (not defined in the configuration alist in `define-servertype!`), then return an empty list.

`serveez-port?` *name* [Scheme Procedure]
Return `#t` if the given string *name* corresponds with a registered port configuration, otherwise `#f`.

`serveez-server?` *name* [Scheme Procedure]
Check whether the given string *name* corresponds with an instantiated server name and return `#t` if so.

`serveez-servertype?` *name* [Scheme Procedure]
Check whether the given string *name* is a valid server type prefix known in Serveez and return `#t` if so. Otherwise return `#f`.

`serveez-exceptions` [*enable*] [Scheme Procedure]
Control the use of exception handlers for the Guile procedure calls of Guile server callbacks. If the optional argument *enable* is `#t`, enable exception handling; if `#f`, disable it. Return the current (boolean) setting.

`serveez-nuke` [*exit-value*] [Scheme Procedure]
Shutdown all network connections and terminate after the next event loop. You should use this instead of calling `quit`. Optional arg *exit-value* specifies an exit value for the serveez program. It is mapped to a number via `scm_exit_value`.

`serveez-loadpath` [*args*] [Scheme Procedure]
Make the search path for the Serveez core library accessible to Scheme. Return a list a each path as previously defined. If *args* is specified, override the current definition of this load path with it. The load path is used to tell Serveez where it can find additional server modules.

`serveez-interfaces` [*args*] [Scheme Procedure]
Make the list of local interfaces accessible to Scheme. Return the local interfaces as a list of ip addresses in dotted decimal form. If *args* are specified, they are added as additional local interfaces.

`getrpc` [*arg*] [Scheme Procedure]
Lookup a network rpc service *arg* (name or service number), and return a network rpc service object. If given no arguments, it behave like `getrpcent`.

`setrpc` [*stayopen*] [Scheme Procedure]
Open and rewind the file `/etc/rpc`. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getrpc`. If *stayopen* is omitted, this is equivalent to calling `endrpcent`. Otherwise it is equivalent to calling `setrpcent` with arg 1.

`portmap` *prognum versnum* [*protocol* [*port*]] [Scheme Procedure]
Establish a (portmap service) mapping between the triple [*prognum*,*versnum*,*protocol*] and *port* on the machine's portmap service. The value of *protocol* is most likely

`IPPROTO_UDP` or `IPPROTO_TCP`. If instead *protocol* and *port* are omitted, destroy all mapping between the triple [*prognum*,*versnum*,*] and ports on the machine's portmap service.

`portmap-list` [*address*] [Scheme Procedure]
Return a list of the current RPC program-to-port mappings on the host located at IP address *address*, which defaults to the local machine's IP address. Return an empty list if either there is no such list available or an error occurred while fetching the list.

`svz:coserver:dns` *host callback* [Scheme Procedure]
Enqueue the *host* string argument into the internal DNS coserver queue. When the coserver responds, the procedure *callback* is run as `(callback addr)`. The *addr* argument passed to the callback is a string representing the appropriate IP address for the given hostname *host*.

`svz:coserver:reverse-dns` *addr callback* [Scheme Procedure]
Enqueue the given *addr* argument, which must be an IP address in network byte order, into the internal reverse DNS coserver queue. When the coserver responds, the procedure *callback* is run as `(callback host)` where *host* is the hostname of the requested IP address *addr*.

`svz:coserver:ident` *sock callback* [Scheme Procedure]
Enqueue the given `#<svz-socket>` *sock* into the internal ident coserver queue. When the coserver responds, it runs the procedure *callback* as `(callback user)`, where *user* is the corresponding username for the client connection *sock*.

3.2.2.6 Callback Prototypes

The Guile interface of Serveez is completely callback driven. Callbacks can be set up in the associative list passed to `define-servertype!`, or by using the predefined procedures described in the previous section. Each of the callbacks is passed certain arguments and is meant to return specific values to indicate success or failure. This section describes each of these callbacks.

`global-init` *servertype* [Scheme Procedure]
This callback is invoked once for every type of server right after the `define-servertype!` statement. Here you can initialise resources shared between all instances of your server type. The callback is optional and can be set up in `define-servertype!`. It should return zero to indicate success and non-zero to indicate failure. If the global initialiser fails, Serveez will refuse to register the server type.

`global-finalize` *servertype* [Scheme Procedure]
If you want to free shared resources, which were possibly allocated within the global initialiser, you can do so here. The callback is invoked when Serveez shuts down (issued by `serveez-nuke`) or the server type gets unregistered for some reason. It should return zero to signal success. The callback can be set up in `define-servertype!` and is optional.

`init` *server* [Scheme Procedure]

Within this callback you can initialise everything you might need for a single instance of your server. The callback is invoked for each server instance which has been created by `define-server!` and should return zero to indicate success, otherwise Serveez rejects the server instance. The callback can be set up in `define-servertype!` and is optional.

`finalize` *server* [Scheme Procedure]

The server instance finaliser gets its instance representation passed as argument. You need to free all resources used by this server instance which might have been allocated within the server instance initialiser or consumed while running. You can set this callback in the `define-servertype!` statement. The callback is optional and should return zero to indicate success.

`detect-proto` *server socket* [Scheme Procedure]

Connection oriented protocols like TCP and PIPE allow more than one server to be listening on the same network port. Therefore, it is necessary to be able to detect the type of client connecting to a port.

This callback takes two arguments; the first is the server instance and the second is the client socket object containing the client connection information. You can set up this callback in the `define-servertype!` statement.

Serveez may invoke this callback several times as data becomes available from the client until one of the servers recognises it. The servers can retrieve the data received so far using the `svz:sock:receive-buffer` call.

To indicate successful client detection, you need to return a non-zero value. (Note that for historical reasons, this is inconsistent with other procedures which return zero on successful completion.) Once the server has indicated success, Serveez invokes any further callbacks for the connection only on that server.

If no server has recognised the client after the first 16 bytes, Serveez will close the connection. The connection will also be closed if the client has not sent enough data for a server to recognise it within 30 seconds of connecting.

If multiple servers are listening on the same network port, Serveez invokes this callback for each of them in an arbitrary order. Only one server at most should indicate successful detection.

This callback is mandatory for servers which get bound to connection oriented protocol (TCP and PIPE) port configurations by `bind-server!`.

`connect-socket` *server socket* [Scheme Procedure]

If the client detection signalled success, this callback is invoked to assign the client connection to a server instance. The arguments are the same as the detection callback. In this callback you can assign all the connection specific callbacks for your server and perform some initial tasks. Basically you should specify the `handle-request` and/or `check-request` callback. This can be achieved by calling `svz:sock:handle-request` and `svz:sock:check-request`. The `connect-socket` callback is also mandatory for connection oriented protocols and must be defined in `define-servertype!`. On success you should return zero, otherwise the connection will be shutdown by Serveez.

`info-server` *server* [Scheme Procedure]
: This callback gets invoked when requested by the builtin Section 3.4.3 [Control Protocol Server], page 42. The callback is optional and can be set up in `define-servertype!`. The returned character string can be multiple lines separated by `\r\n` (but without a trailing separator). Usually you will return information about the server instance configuration and/or state.

`info-client` *server socket* [Scheme Procedure]
: This callback is optional. You can set it up in the `define-servertype!` procedure. It is meant to provide socket structure specific information. (The socket structure is a client/child of the given server instance.) You need to return a single line character string without trailing newlines. The information provided can be requested by the builtin Section 3.4.3 [Control Protocol Server], page 42.

`notify` *server* [Scheme Procedure]
: The server instance notifier callback will be invoked whenever there is idle time available. In practice, it is run approximately once a second. A server instance can use it to perform periodic tasks. The callback is optional and can be set up in `define-servertype!`.

`reset` *server* [Scheme Procedure]
: This callback is invoked when the Serveez process receives a `SIGHUP` signal which can be issued via '`killall -HUP serveez`' from user land. If the underlying operating system does not provide `SIGHUP` there is no use for this callback. It provides the possibility to perform asynchronous tasks scheduled from outside Serveez. You can optionally set it up in the `define-servertype!` procedure.

`handle-request` *socket binary size* [Scheme Procedure]
: This callback is invoked whenever a complete packet has been detected in the receive buffer. The packet data is passed to the callback as a `#<svz-binary>`. The *size* argument is passed for convenience and specifies the length of the packet in bytes.

  The detection, and therefore the invocation, can be made in one of two ways. When Serveez can determine itself when a packet is complete, the callback will be invoked directly. Serveez can make this determination for connections with packet oriented protocols such as UDP and ICMP, or if you tell Serveez how to parse the packet using `svz:sock:boundary sock delimiter` or `svz:sock:boundary sock size` and do not specify a `check-request` callback.

  Whenever you specify a `check-request` callback to determine when a packet is complete, it becomes the responsiblity of that callback to invoke `handle-request` itself.

  Serveez recognises two different return value meanings. For connection oriented protocols (TCP and PIPE), zero indicates success and non-zero failure; on failure, Serveez will shutdown the connection. For packet oriented protocols (UDP and ICMP), a non-zero return value indicates that your server was able to process the passed packet data, otherwise (zero return value) the packet can be passed to other servers listening on the same port configuration.

  This callback must be specified in `define-servertype!` for packet oriented protocols (UDP and ICMP) but is optional otherwise. You can modify the callback by calling `svz:sock:handle-request`.

`check-request` *socket* [Scheme Procedure]

This callback is invoked whenever new data has arrived in the receive buffer. The receive buffer of the given `#<svz-socket>` can be obtained using `svz:sock:receive-buffer`. The callback is initially not set and can be set up with `svz:sock:check-request`. Its purpose is to check whether a complete request was received. If so, it should be handled (by running the `handle-request` callback) and removed from the receive buffer (using `svz:sock:receive-buffer-reduce`). The callback is for connection oriented protocols (TCP and PIPE) only. You should return zero to indicate success and non-zero to indicate failure. On failure Serveez shuts the connection down.

`disconnected` *socket* [Scheme Procedure]

The `disconnected` callback gets invoked whenever the socket is lost for some external reason and is going to be shutdown by Serveez. It can be set up with `svz:sock:disconnected`.

`kicked` *socket reason* [Scheme Procedure]

This callback gets invoked whenever the socket gets closed by Serveez intentionally. It can be set up with `svz:sock:kicked`. The *reason* argument can be either `KICK_FLOOD`, indicating the socket is a victim of the builtin flood protection, or `KICK_QUEUE` which indicates a send buffer overflow.

`idle` *socket* [Scheme Procedure]

The `idle` callback gets invoked from the periodic task scheduler, which maintains a `idle-counter` for each socket structure. This counter is decremented whenever Serveez becomes idle and the callback is invoked when it drops to zero. The `idle` callback can set its socket's `idle-counter` to some value with the procedure `svz:sock:idle-counter` and thus re-schedule itself for a later task. You can set up this callback with `svz:sock:idle`.

`trigger-condition` *socket* [Scheme Procedure]

This callback is invoked once every server loop for the socket structure. If you return `#f` nothing else is happening. Otherwise the `trigger` callback will be invoked immediately. You can set up the callback using the procedure `svz:sock:trigger-condition`.

`trigger` *socket* [Scheme Procedure]

The `trigger` callback is invoked when the `trigger-condition` returns `#t`. The callback can be set up with the procedure `svz:sock:trigger`. Returning a non-zero value shuts the connection down. A zero return value indicates success. This callback can be used to perform connection related updates, e.g., you can ensure a certain send buffer fill.

`check-oob-request` *socket oob-byte* [Scheme Procedure]

This callback is invoked whenever urgent data (out-of-band) has been detected on a socket. Initially this event is ignored and the callback can be set up with the procedure `svz:sock:check-oob-request`. The *oob-byte* argument is a number containing the received out-of-band data byte ranging from 0 to 255. If the callback returns non-zero

the connection will be shutdown. A zero return value indicates success. You can use `svz:sock:send-oob` to send a single out-of-band data byte.

Please note: The urgent data is not supported by all operating systems. Also it does not work for all types of network protocols. We verified it to be working for TCP streams on GNU/Linux 2.x.x and Windows 95; let us know if/how it works on other platforms.

3.2.3 Builtin servers

All of the servers listed in Section 3.4 [Existing servers], page 36, are builtin servers. The following sections describe in detail how to setup a new builtin server type. This kind of server will be part of the Serveez executable. That is why you should make it configurable in the configure script via a '`--enable-xxxserver`' argument.

3.2.3.1 Making and configuring preparations

Serveez is configured and built via automake and autoconf. That is why you are not supposed to write your own Makefiles but simplified `Makefile.am`s. Automake will automatically generate dependencies and compiler/linker command lines. Here are the steps you basically need to follow:

- Change to the `src/` directory in the source tree.
- Edit the `Makefile.am`. Add your sub directory name and library name which you are going to create.
- Now create the sub directory and change into it.
- You need to create a new `Makefile.am` therein. If you want to have this file `configure`d you need to add a further line to the `AC_OUTPUT` statement in `configure.ac` which is in the top level directory. You have to put at least the following into the newly created `Makefile.am`:

```
noinst_LIBRARIES = libfoo.a
libfoo_a_SOURCES = foo-proto.h foo-proto.c
INCLUDES = $(SERVEEZ_CFLAGS) -I$(top_srcdir)/src
CLEANFILES = *~
MAINTAINERCLEANFILES = Makefile.in
```

- Just have a look at all the other server directories. For more information about automake read the info pages.

3.2.3.2 Server header file `foo-proto.h`

This file contains at least your server's `extern` declaration of your server definition which must be available from the outside. The `foo` server implements all kinds of configuration items which can be integers, integer arrays, strings, string arrays, port configurations, booleans and hash maps. Every item of the server configuration can later be manipulated from the configuration file.

3.2.3.3 Server implementation file `foo-proto.c`

If you want to define default values for your configuration you have to define them somewhere and put them into the default configuration structure. This structure will be used to instantiate your server. For this example we simply called it simply `foo_config`.

In order to associate the configuration items in a server configuration to keywords within the configuration file you have to define an array of key-value-pairs. This is done in the `foo_config_prototype` field. There are several macros which make different associations. These are the `SVZ_REGISTER_*` macros which take three arguments. The first argument is the keyword which will occur in the configuration file, the second is the associated item in your default configuration structure and the last argument specifies if this item is defaultable or not.

3.2.3.4 Server definition

The server definition is in a way the '`class`' of your server. Together with the default values (`foo_config_prototype`) it serves as a template for newly instantiated servers. The structure contains a long and a short description of your server. The short name is used as the prefix for all server instances of this specific type. The long description is used in the control protocol (See Section 3.4.3 [Control Protocol Server], page 42.). The server definition also contains the callbacks your server (mandatorily) provides.

3.2.3.5 Server callbacks

There are several callback routines, which get called in order to instantiate the server and for describing the actual behaviour of your server. Here are the description of all of these callbacks. Some of them have to be implemented. Others have reasonable default values.

`global initializer (optional)`
: This callback is executed once for every type of server. Here you can initialize data or whatever is shared between all instances of your server. For instance the HTTP server initializes its file cache here.

`global finalizer (optional)`
: If you want to free shared resources which were possibly allocated within the global initializer you can do so here. The foo server frees its default hash previously allocated in the global initializer.

`instance initializer (mandatory)`
: Within this routine you can initialize everything you might need for one instance of your server. The foo server does not do anything in this callback.

`instance finalizer (optional)`
: The server instance finalizer gets its instance representation as argument. You have to free all resources used by this server instance.

`protocol detection (mandatory)`
: Because it is possible to have more than one server listening on one network port we need to detect the type of client which is connecting to this port. The foo server checks if the first five bytes the client was sending is identifying it as a foo client. This routine is getting two arguments where the first one is a pointer to this servers instance and the second is the client socket object containing all information of the client connection. This structure is described a bit later. Be patient. For successful client detection return non-zero value.

`socket connection (mandatory)`
: If the client detection signaled success this routine is called to assign the client connection to the server instance. The arguments are just the same as in the

detection routine. In this callback you can assign all the connection specific callbacks for your server and do some initial things. The foo server sets the `check_request` callback to the default `svz_sock_check_request` which is using the packet delimiter information to find whole packets. When a client sent such a packet the `handle_request` callback is executed. That is why the foo server assigns the `handle_request` method.

`client info (optional)`
: If this callback is given the control protocol (See Section 3.4.3 [Control Protocol Server], page 42.) can give information about a specific client if requested with '`stat id NUM`'. The first argument given is the server instance and the second one the client's socket structure. You have to return a static single line character string.

`server info (optional)`
: This function is called when listing the server instances via '`stat all`' from the control protocol (See Section 3.4.3 [Control Protocol Server], page 42.). The returned character string might be multilined separated by `\r\n` (no trailing separator). Usually you will return all the server configuration information.

`notifier (optional)`
: If this callback is not `NULL` it is called whenever there is some time left. It gets the server instance itself as argument. Actually it gets called every second.

`handle request (mandatory for UDP and ICMP servers)`
: The arguments to this callback are the client's socket structure, the address of the packet data and its length. When implementing a UDP or ICMP server you need to return non-zero if your server could process the packet. Thus it is possible that there are multiple UDP servers on a single port.

3.2.3.6 Make your server available

You distribute your server by editing the `cfgfile.c` file in the `src/` directory. There you have to include the servers header file and add the server definition by calling `svz_servertype_add`

3.2.3.7 More detailed description of the callback system and structures

The client connection information is stored within the `svz_socket_t` object. All of the client connection specific callbacks get this object as first argument. Following is a description of the most important elements of this object.

`int id` The socket id is a unique id for a client connection.

`int version`
: This item validates this socket structure. If you pass the `id` and `version` to a coserver you can check if the delivered socket structure is the original or not within the coserver callback.

`int proto` The `proto` flag determines a server sockets protocol type which can be `PROTO_PIPE`, `PROTO_TCP`, `PROTO_UDP`, `PROTO_ICMP` or `PROTO_RAW`.

`int flags` The flag field of the client connection contains informations about the state of this connection. See `socket.h` in the `src/libserveez/` directory for more information. Basically this bitfield specifies how this object is handled by the main server loop.

`int userflags`
This bitfield could be used for protocol specific information. You can use it for any information.

`char *boundary, int boundary_size`
If you are going to write a packet oriented protocol server you can use the `svz_sock_check_request` method to parse packets. These two properties describe the packet delimiter.

`char *send_buffer, int send_buffer_size, int send_buffer_fill`
This is the outgoing data for a client connection object.

`char *recv_buffer, int recv_buffer_size, int recv_buffer_fill`
Within the receive buffer all incoming data for a connection object is stored. This buffer is at least used for the client detection callback.

`int read_socket (svz_socket_t)`
This callback gets called whenever data is available on the socket. Normally, this is set to a default function which reads all available data from the socket and feeds it to `check_request`, but specific sockets may need other policies.

`int write_socket (svz_socket_t)`
This routine is called when data is is valid in the output buffer and the socket gets available for writing. You normally leave this callback untouched. It simply writes as much data as possible to the socket and removes the data from the send buffer.

`int disconnected_socket (svz_socket_t)`
This gets called whenever the socket is lost for some external reason.

`int connected_socket (svz_socket_t)`
If some piece of code tries to connect to another host via `svz_tcp_connect` this connection might be established some time later. This callback gets called when the socket is finally connected.

`int kicked_socket (svz_socket_t, int)`
We call this whenever the socket gets closed by us. The second argument specifies a reason.

`int check_request (svz_socket_t)`
This gets called whenever data was read from the socket. Its purpose is to check whether a complete request was read, and if it was, it should be handled and removed from the input buffer.

`int handle_request (svz_socket_t, char *, int)`
This gets called when the `check_request` got a valid packet. The request arguments contains the actual packet and the second argument is the length of this packet including the packet delimiter.

`int idle_func (svz_socket_t)`
: This callback gets called from the periodic task scheduler. Whenever `idle_counter` (see below) is non-zero, it is decremented and `idle_func` gets called when it drops to zero. `idle_func` can reset `idle_counter` to some value and thus can re-schedule itself for a later task.

`int idle_counter`
: Counter for calls to `idle_func`.

`void *data`
: Miscellaneous field. Listener keeps array of server instances here. This array is `NULL` terminated. Some servers store server specific information here.

`void *cfg` When the final protocol detection has been done `cfg` should contain a pointer to the actual configuration hash map taken from the server instance object.

3.2.3.8 Using coservers

Coservers are designed to complete blocking tasks. Each coserver runs in its own thread/process. There are several coservers implemented: the dns, reverse dns and ident coserver. You need to implement the callback which gets called when a coserver completed its task. This routine must be a `svz_coserver_handle_result_t`. The first argument is the actual coserver result which might be `NULL` if the request could not be fulfilled and the latter two arguments are the arguments you specified yourself when issuing the request. To invoke a coserver you use one of the `svz_coserver_*` macros. The foo server uses the reverse dns coserver to identify the host name of the remote client.

3.3 Some words about server configuration

If you define a server you basically pass an instance name and a list of items to the `define-server!` procedure. Each item has a name and a value. A value has a type. We provide several types: integers (numbers), integer arrays, strings, string arrays, booleans (yes/no-values), hashes (associations) and port configurations.

The following table shows how each kind of value is set up in the configuration file. *item* is the name of the item to be configured.

Integer
: Example: (item . 42)

Integer array
: Example: (item . (0 1 2 3))

String
: Example: (item . "a character string")

String array
: Example: (item . ("abc" "cba" "bca" "acb"))

Boolean
: A normal boolean in guile is represented by #t or #f. But the configuration

file parser additional understand some bare words and numbers.
Example: (item . #f)

Hash

Hash maps associate keys with values. Both must be character strings.
Example: (item . (key1 . "value1") (key2 . "value2"))

Port configuration

See Section 1.5.1 [Define ports], page 6, for more information on this. When configuring a port configuration you need to define it via `define-port!` previously and put its symbolic name into the configuration.
Example: (item . foo-tcp-port)

The next chapter describes the servers currently implemented using Serveez. The configuration items used by each of them are described in the following format:

`NameOfTheItem (Type, default: DefaultValue, Comments)`

Description of the configuration item named '`NameOfTheItem`' (case sensitive). '`Type`' can be either '`integer`', '`integer array`', '`string`', '`string array`', '`boolean`', '`hash`' or '`port configuration`'. The '`Comments`' is an optional text.

The example configuration file `data/serveez.cfg` contains an example definition of each server already implemented. You can copy and modify the example for an easy start.

3.4 Existing servers

3.4.1 HTTP Server

3.4.1.1 General description

The integrated HTTP server was originally meant to be a simple but fast document server. But now it can even execute CGI scripts. The GET, HEAD and POST methods are fully functional. Additionally Serveez produces directory listings when no standard document file (e.g., `index.html`) has been found at the requested document node (directory). Furthermore it implements a file cache for speeding up repetitive HTTP request.

In comparison to other web server projects like Apache and Roxen this web server is really fast. Comparative benchmarks will follow. The benchmark system is a 233 MHz Mobile Pentium MMX. Both the server and the client (http_load - multiprocessing http test client) ran on the same computer.

`Small files`

The small-file test load consists of 1000 files, each 1KB long, requested randomly.

```
concurrent fetches   1   10   50  100  200  500  1000
hits/second        501  520  481  475  420  390   295
```

`CGI` The CGI test load consists of a trivial "hello world" C program. I noticed GNU/Linux (2.2.17 in this case, probably others too) to throw "Resource temporarily unavailable" errors when `fork`ing very fast. This limits the test to about 200 concurrent fetches on the test system.

`Large files`
: The large-file test load consists of 100 files, each 1MB long, requested randomly. Also, each connection is throttled to simulate a 33.6Kbps modem. Note that 1000 33.6Kbps connections is 3/4 of a T3. There was no problem to get 1000+ concurrent fetches.

3.4.1.2 Configuration

The following options can be set from the configuration file.

`indexfile (string, default: index.html)`
: The `indexfile` parameter is the default file served by the HTTP server when the user does not specify a file but a document node (e.g., `http://www.lkcc.org/`).

`docs (string, default: ../show)`
: The `docs` parameter is the document root where the server finds its web documents.

`userdir (string, default: public_html)`
: Each '`~user`' request gets converted into the given users home directory. The string will be appended to this directory. Its default value is '`public_html`'.

`cgi-url (string, default: /cgi-bin)`
: This parameter is the first part of the URL the HTTP server identifies a CGI request. For instance if you specify here `/cgi-bin` and the user requests `http://www.lkcc.org/cgi-bin/test.pl` then the HTTP server tries to execute the program `test.pl` within the `cgi-dir` (see below) and pipes its output to the user.

`cgi-dir (string, default: ./cgibin)`
: The `cgi-dir` is the CGI document root (on the server).

`cgi-application (hash, default: empty)`
: Within the MinGW32 port you can use this hash to associate certain file suffices with applications on your computer (e.g., `pl` with `perl`). This is necessary because there is no possibility to check whether a file is executable on Win32.

`cache-size (integer, default: 200 kb)`
: This specifies the size of the document cache in bytes for each cache entry.

`cache-entries (integer, default: 64)`
: This parameter specifies the maximum number of HTTP file cache entries (files). When you instantiate more than one HTTP server the biggest value wins. The HTTP file cache is shared by all HTTP servers.
Please note: If your harddrive/filesystem combination proves to be faster than the HTTP file cache you should disable it by setting both `cache-size` and `cache-entries` to zero.

`timeout (integer, default: 15)`
: The `timeout` value is the amount of time in seconds after which a keep-alive connection (this is a HTTP/1.1 feature) will be closed when it has been idle.

`keepalive (integer, default: 10)`
: On one keep-alive connection can be served the number of `keepalive` documents at all. Then the connection will be closed. Both this and the `timeout` value are just to be on the safe side. They protect against idle and high traffic connections.

`default-type (string, default: text/plain)`
: The `default-type` is the default content type the HTTP server assumes if it can not identify a served file by the `types` hash and the `type-file` (see below).

`type-file (string, default: /etc/mime.types)`
: This should be a file like the `/etc/mime.types` on Unix systems. It associates file suffices with MIME types.

`types (hash, default: empty)`
: If you want to specify special content types do it here. This parameter is a hash map associating file suffices with HTTP content types (MIME types).

`admin (string, default: root@localhost)`
: Your address, where problems with the server should be e-mailed. This address appears on some server-generated pages, such as error documents.

`host (string, default: localhost)`
: This is the host name of your web server. Sometimes the server has to send back its own name to the client. It will use this value. Be aware that you cannot invent such a name.

`nslookup (boolean, default: false)`
: If this is true the HTTP server invokes a reverse DNS lookup for each client connection in order to replace the remote ip address with the remote host name in the access logfile.

`ident (boolean, default: false)`
: If this is true the HTTP server processes identd requests for each client connection for logging purposes.

`logfile (string, default: http-access.log)`
: The location of the access logfile. For each HTTP request a line gets appended to this file.

`logformat (string, default: CLF)`
: The format of the access logfile. There are special placeholders for different kinds of logging information. The default log format is the Common Log Format (CLF). It contains a separate line for each request. A line is composed of several tokens separated by spaces.

```
CLF = host ident authuser date request status bytes
```

: If a token does not have a value then it is represented by a hyphen (-). The meanings and values of these tokens are as follows:

`%h (host)`
: The fully-qualified domain name of the client, or its IP number if the name is not available.

`%i (ident)`
: This is the identity information reported by the client. Not active, so we will see a hyphen (-).

`%u (authuser)`
: If the request was for an password protected document, then this is the userid used in the request.

`%t (date)` The date and time of the request, in the following format:

```
date   = [day/month/year:hour:minute:second zone]
day    = 2*digit
month  = 3*letter
year   = 4*digit
hour   = 2*digit
minute = 2*digit
second = 2*digit
zone   = (‘+’ | ‘-’) 4*digit
```

`%R (request)`
: The request line from the client, enclosed in double quotes (").

`%r (referrer)`
: Which document referred to this document.

`%a (agent)`
: What kind of web browser did the remote client use.

`%c (status)`
: The three digit status code returned to the client.

`%l (bytes)`
: The number of bytes in the object returned to the client, not including any headers.

3.4.2 IRC Server

3.4.2.1 General description

Internet Relay Chat. The mother of all chat systems. The integrated IRC server is intended to be compatible with the EFNet. There are no good possibilities to test this in real life, so it is still under heavy construction. But it can be used as a standalone server anyway.

IRC itself is a teleconferencing system, which (through the use of the client-server model) is well-suited for running on many machines in a distributed fashion. A typical setup involves a single process (the server) forming a central point for clients (or other servers) to connect to, performing the required message delivery/multiplexing and other functions.

The server forms the backbone of IRC, providing a point for clients and servers to connect to. Clients connect to talk to each other. Servers connect to build up a network of servers. IRC server connections have to build up a spanning tree. Loops are not allowed. Each server acts as a central node for the rest of the network it sees.

3.4.2.2 Configuration

The following table shows the configuration keys provided. Most of the configuration items are similar to those of an Hybrid IRC server. They seem archaic at first sight but IRC operators are used to it. Refer to the Hybrid documentation for further information. It can be found on the EFNet web page.

`MOTD-file (string, default: ../data/irc-MOTD.txt)`
: When a user initially joins it will get this file's content as the message of the day comment. When changing on disk the server will notice that and reload the file automatically.

`INFO-file (string, default: no file)`
: The `INFO-file`s content gets displayed when the user issues the /INFO command.

`tsdelta (integer, default: 0)`
: This value is the timestamp delta value to UTC (Coordinated Universal Time) in seconds.

`channels-per-user (integer, default: 10)`
: Configures the maximum number of channels a single local user can join.

`admininfo (string, no default)`
: Some administrative information delivered on the /ADMIN command.

`M-line (string, no default, mandatory)`
: The TCP level configuration of this IRC server. The server info field is sometimes given to the client for informational use. The server will croak about if the settings do not correspond with the actual bindings. The format of this line is:

```
":" virtual hostname
":" optional bind address (real hostname)
":" server info: "World's best IRC server"
":" port
```

`A-line (string, no default, mandatory)`
: The administrative info, printed by the /ADMIN command.

```
":" administrative info (department, university)
":" the server's geographical location
":" email address for a person responsible for the IRC server
```

`Y-lines (string array, no default, suggested)`
: The connection classes. They are used in other parameters (e.g., I-lines). A Y-line describes a group of connections. You usually have at least two Y-lines: One for server connections and one for client connections. Format of each line is:

```
":" class number (higher numbers refer to a higher priority)
":" ping frequency (in seconds)
":" connect frequency in seconds for servers, 0 for
    client classes
```

```
":" maximum number of links in this class
":" send queue size
```

`I-lines (string array, no default, mandatory)`
: Authorization of clients, wildcards permitted, a valid client is matched `user@ip` OR `user@host`.

```
":" user@ip, you can specify 'NOMATCH' here to force
    matching user@host
":" password (optional)
":" user@host
":" password (optional)
":" connection class number (YLine)
```

`O-lines (string array, no default, optional)`
: Authorize operator, wildcards allowed.

```
":" user@host, user@ forces checking ident
":" password
":" nick
```

`o-lines (string array, no default, optional)`
: Authorize local operator.

```
":" user@host, user@ forces checking ident
":" password
":" nick
```

`C-lines (string array, no default, networked)`
: List of servers to connect to. Note: C and N lines can also use the user@ combination in order to check specific users (ident) starting servers. C and N lines are usually given in pairs.

```
":" host name
":" password
":" server name (virtual)
":" port (if not given we will not connect)
":" connection class number (YLine)
```

`N-lines (string array, no default, networked)`
: Servers which may connect.

```
":" host name
":" password
":" server name (virtual host name)
":" password
":" how many components of your own server's name to strip
    off the front and be replaced with a '*'.
":" connection class number (YLine)
```

`K-lines (string array, no default, optional)`
: Kill user, wildcards allowed.

```
":" host
":" time of day
":" user
```

3.4.3 Control Protocol Server

If the GNU Serveez package is configured with the control protocol enabled, running '`serveez --help`' will show the option `-P` and the following documentation applies. Otherwise, feel free to skip to the next section.

3.4.3.1 General description

Serveez implements something like a telnet protocol for administrative purposes. You just need to start a telnet session like:

```
$ telnet www.lkcc.org 42420
```

After pressing `RET` you will be asked for a password which you might setup passing Serveez the -P argument. See Chapter 1 [Using Serveez], page 1. The next section describes the interactive commands available.

3.4.3.2 Using the Control Protocol

'`help`' This command will give you a very short help screen of all available commands.

'`quit`' This command closes the connection to Serveez.

'`restart ident`'
Restarts the internal ident coserver. This is useful if you just want to start a new one if the old one died or is otherwise unusable.

'`restart dns`'
Restarts the internal dns lookup server.

'`restart reverse dns`'
Restarts the internal reverse dns lookup server.

'`killall`' This might be useful if Serveez seems to be unstable but you do not want to restart it. With '`killall`' you disconnect all client network connections except the control protocol connections.

'`kill id NUM`'
Disconnects a specific connection identified by its ID. These IDs will be stated when you type '`stat con`' (see below).

'`stat`' General statistics about Serveez. This will show you some useful information about the computer Serveez is running on and about the state of Serveez in general.

'`stat coserver`'
Statistics about all running coserver instances.

'`stat SERVER`'
This command is for selecting certain server instances to be listed. SERVER is one of server names you specified in the configuration file.

'`stat id NUM`'
Show statistics about a specific connection. This will give you all available information about every connection you specified. See Section 3.2 [Writing servers], page 18, for more information about how to provide these information.

'`stat con`' Connection statistics. This will give a list of all socket structures within Serveez. If you want more detailed information about specific connections, coservers or servers you need to request these information with '`stat id NUM`' or '`stat all`'.

'`stat all`' Server and coserver instance statistics. This command lists all the information about instantiated servers and coservers. See Section 3.2 [Writing servers], page 18, for more information about how to provide these information.

'`stat cache`'
HTTP cache statistics. This command produces an output something like the following where '`File`' is the short name of the cache entry, '`Size`' the cache size, '`Usage`' the amount of connections currently using this entry, '`Hits`' the amount of cache hits, '`Recent`' the cache strategy flag (newer entries have larger numbers) and '`Ready`' is the current state of the cache entry.

```
File                       Size  Usage  Hits Recent Ready
zlib-1.1.3-20000531.zip   45393      0     0      1 Yes
texinfo.tex              200531      0     0      2 Yes
shayne.txt                 2534      0     1      1 Yes

Total : 248458 byte in 3 cache entries
```

'`kill cache`'
Reinitialize the HTTP file cache. Flushes all files from the cache.

3.4.3.3 Configuration

There is nothing to be configured yet.

3.4.4 Foo Server

3.4.4.1 General description

The Foo Server is a simple example on how to write Internet protocol servers with Serveez. See Section 3.2 [Writing servers], page 18.

3.4.4.2 Configuration

There are all kinds of configuration items. They are used to explain the implementation of servers. A complete list will follow.

`port (port configuration, default: tcp, 42421, *)`
Sets up the TCP port and local address.

`bar (integer, no default)`
Some integer value. Printed as server information.

`reply (string, default: Default reply)`
Some string. Printed as server information.

`messages (string array, default: ...)`
Some string array which is actually a list of strings. Also printed as server information.

`ports (integer array, default: 1, 2, 3, 4)`
Some array of integer numbers. Printed as server information.

`assoc (hash, default, default: ...)`
: An hash map associating keys with values. Printed as server information.

`truth (boolean, default: true)`
: Some boolean value. Printed as server information.

3.4.5 SNTP Server

3.4.5.1 General

The SNTP server can be queried with the '`netdate`' command. It is used to synchronize time and dates between Internet hosts. The protocol is described in the ARPA Internet RFC 868. Thus it is not really an SNTP server as described by RFC 2030 (Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI). It is rather an excellent example on how to implement a UDP server in Serveez.

This protocol provides a site-independent, machine readable date and time. The Time service sends back time in seconds since midnight on January first 1900.

One motivation arises from the fact that not all systems have a date/time clock, and all are subject to occasional human or machine error. The use of time-servers makes it possible to quickly confirm or correct a system's idea of the time, by making a brief poll of several independent sites on the network.

3.4.5.2 Configuration

The configuration of this server does not require any item.

3.4.6 Gnutella Spider

3.4.6.1 What is it ?

The Gnutella net is a peer-to-peer network which is based on client programs only. There are no servers. The network itself is formed by client connections only. Generally the Gnutella network is for sharing files of any kind.

This Gnutella spider is for seeking the needle in the haystack. Once connected to the network it regularly tries to find certain files in there. It keeps track of all connected clients and tries to reconnect them if the current connections are lost.

Gnutella, however has nothing to do with the GNU project. The original client is just a free (as in free beer) piece of software. With Serveez you have a free (as in freedom) way to use it. Have a look at the Gnutella page for further information.

3.4.6.2 Configuration

The Gnutella spider knows the following configurations items.

`net-url (string, default: gnutella-net)`
: If you want to see the host catcher list of this Gnutella spider you can connect to this port with any WWW browser at `http://host:port/net-url`. The `host:port` combinations depend on the bindings.

`hosts (string array, no default)`
: This is the start of the haystack, the initial host list of the clients the spider tries to connect to. Each list item should be of the format `ip:port` (e.g.,

'146.145.85.34:6346'). You can also pass Internet host names. If the port information is left blank it defaults to 6346. If you need some entry point for the Gnutella network have a look at http://www.gnutellahosts.com/ or http://www.gnutellanet.com/.

search (string array, default: Puppe3000, Meret Becker)
: This is the needle. Each search line is either a set of space delimited tokens where every token must match. Or a kind of wildcard expression including '?' and '*'. Search lines are always matched case insensitive.

search-limit (integer, default: 30)
: This limits how many results the Gnutella spider returns to other people searching your files. This is for protection against * search requests.

max-ttl (integer, default: 5)
: Every Gnutella packet has got a TTL. This is the maximum TTL allowed for outgoing packets. When a packet comes in it gets its TTL value decremented and is forwarded to it destination. If however an incoming packet has a TTL larger than max-ttl the ttl value is set to max-ttl. This is necessary since most people use far too large TTL values.

ttl (integer, default: 5)
: When creating a new Gnutella packet we use this as TTL. Please use a sane value. This ttl needs not to be as large as it is for IP packets. A value below 10 is more than enough. Have a look at the Gnutella page for a calculation of a 'sane value'.

download-path (string, default: /tmp)
: This is where the spider saves needles in.

share-path (string, default: /tmp)
: Here are all the files we share with others. The Gnutella spider will recurse into directories. So be careful with this option.

max-downloads (integer, default: 4)
: Maximum number of concurrent downloads from the network.

max-uploads (integer, default: 4)
: Maximum number of concurrent uploads to the network.

connection-speed (integer, default: 28)
: This is what we send as our connection speed in KBit/s. We also use this value to throttle down the network transfer rate for file uploads.

min-speed (integer, default: 28)
: Search for needles on hosts with a minimum speed. Set it to 0 if you do not care about that. This value is in KBit/s, too.

file-extensions (string array, default: empty list)
: If we get replies on search queries we check if the file extension of this reply matches one of these extensions. Useful extensions are 'mp3' and 'mpg'.

`connections (integer, default: 4)`
: This is the number of concurrent connections the Gnutella spider tries to keep up to the network. The IP addresses and the port information is taken from the host catcher hash.

`force-ip (string, default: not set)`
: You can force the Gnutella spider to send outgoing replies with this IP as host information. Must be in dotted decimals notation. This is useful if you are behind a masquerading gateway. You need to install some kind of port forwarder on the gateway so other people can reach you from the outside. Serveez is a good port forwarder.

`force-port (integer, default: not set)`
: Force the Gnutella spider to send outgoing replies with the `force-port` as port information. See above for more information.

`disable (boolean, default: false)`
: With this configuration option you can disable the bindings for a Gnutella server instance. This means that no remote client can connect without being told so (e.g., by push requests).

3.4.7 Tunnel Server

3.4.7.1 General description

The Tunnel server is for mapping one port configuration to another. So we should rather speak of a port forwarder. Two port forwarders can form a tunnel. Generally this means that you can setup Serveez to accept network or pipe connections in order to pass all transfer data on this line to another port configuration. This can be useful to work around gateways and firewalls. When instantiating an ICMP source or destination you must ensure root privileges for the application. On Windows NT and Windows 2000 you need to be either logged in as Administrator or have set the registry key HKLM\System\CurrentControlSet\Services\Afd\Parameters\DisableRawSecurity to 1 (DWORD). One of the given examples in `serveez.cfg` shows how you can setup a tunnel server for forwarding a pipe connection. Please keep in mind when forwarding a TCP or pipe connection over ICMP or UDP you loose reliability since the latter two are packet oriented rather than connection oriented. We are not willing to implement our own TCP stack to work on ICMP/UDP directly.

Forwarding between the same types of connection is always possible. When forwarding to an ICMP tunnel we use a special protocol which we will outline in the following section.

3.4.7.2 Extended ICMP protocol specification

Since ICMP (Internet Control Message Protocol) does have a fixed packet format we had to extend it in order to use it for our own purposes. The protocol field of the IP header contains a binary '`1`' which is the identifier for ICMP (e.g., '`6`' identifies TCP). When creating an ICMP socket the IP header is always generated by the kernel. This is the main difference to raw sockets where the IP header must be generated at userspace level.

When receiving an ICMP packet it also contains the IP header. When sending an ICMP packet you must not prepend this IP header. The kernel will do this itself. The IP header

always follows the actual ICMP header followed by the ICMP packet data. Since this section does not cover raw sockets we leave the IP header structure out here.

The modified ICMP message format is as:

Offset	Size	Meaning
0	1	Message type.
1	1	Message type sub code.
2	2	Checksum.
4	2	Senders unique identifier.
6	2	Sequence number.
8	2	Port number.
10	0 - 65506	Packet load.

Each of these fields can be modified and processed by Serveez and do not get touched by the kernel at all. The ICMP socket implementation of Serveez differentiates two types of sockets: listening and connected ICMP sockets. This is non-standard because it actually makes no sense since there is no difference for the kernel. The introduction of these semantics allow Serveez to forward data between connection-oriented (TCP and named pipes) and packet-oriented (UDP and ICMP) protocols.

Message type

Valid message types are for instance '`8`' for an echo message and '`0`' for its echo reply. These two messages are used for the systems builtin ping services. Serveez uses its own message type identifier which is '`42`' (`ICMP_SERVEEZ`) by default.

Message type sub code

Serveez also defines its own message type sub codes described in the following table.

Sub code	Constant identifier	Description
0	`ICMP_SERVEEZ_DATA`	packet contains data
1	`ICMP_SERVEEZ_REQ`	unused
2	`ICMP_SERVEEZ_ACK`	unused
3	`ICMP_SERVEEZ_CLOSE`	disconnection message
4	`ICMP_SERVEEZ_CONNECT`	connect message

Checksum The checksum field of the ICMP header is used to check the ICMP headers and the payloads (packet data) validity. We are using the standard Internet Checksum algorithm described in RFC 1071. If the check failed we drop the packet.

Senders unique identifier

The senders identifier field is used to determine if a received packet has been sent by the sender itself and should therefore be dropped. This happens because each ICMP socket setup for receiving gets all sent ICMP packets system wide. Thus Serveez will even be notified if the kernel creates some echo reply or destination unreachable message due to a request completely outside the scope of Serveez.

Sequence number
: Each connected ICMP socket increments its sequence number when sending a packet. Thus a connection message type packet of such a socket always has a zero sequence number. This field could (but is not yet) also be used to reorder ICMP packets or to detect missing packets.

Port number
: The port field of the modified packet format helps Serveez to establish connected ICMP sockets. A simple packet filter detects if a received packet is kind of reply to a sockets sent packets by comparing this port number. The packet is dropped if the comparison fails and it is not a listening socket.

Except the data message type subcode all ICMP packets created and sent by Serveez have a zero payload. The connect message subcode identifies a new connection and the disconnection message subcode its shutdown without actually transmitting data. These two subcodes emulate a TCP connections `connect`, `accept` and `shutdown` system call.

3.4.7.3 Configuration

This might be the most easiest configuration to setup. You essentially need to define the source port configuration and the target port configuration. The `serveez.cfg` in the `data/` directory shows two example configurations how to tunnel TCP connections over UDP and ICMP. The UDP tunnel accesses the standard HTTP port 80 and the ICMP tunnel accesses the standard Telnet port 23.

`source (port configuration, no default)`
: The source port configuration. This is usually the same you bind the server to.

`target (port configuration, no default)`
: The target port configuration.

3.4.8 Fake Ident Server

3.4.8.1 General description

Most systems run the 'ident protocol' on port 113. Internet hosts can connect to that port and find out what user is having a connection to the host. For example a webserver can query your username when you fetch a file (e.g., Serveez' internal ident-coserver can do that). Most IRC servers protect themselves by allowing only users that have a valid ident response. Therefore mIRC (for windoze) has a built in ident server. This fake ident server can be used to 'fake' a response. This is useful when you connect through a masquerading gateway and the gateway cannot handle ident requests correctly. (Or, of course, you are using windoze, need an ident response and do not have mIRC at hand.)

This server has two modes of operation. In one mode all requests get '`ERROR : NO-USER`' as response. This is a valid but not very helpful response. The other mode makes the server send a valid and useful response. It contains a system type and a username. The system type is usually 'UNIX'. Others are valid but never used (at least i have never seen something else).

3.4.8.2 Configuration

This server is easy to configure.

`systemtype (string, default: UNIX)`
: The system type to respond. The username field of the response has other meanings depending on this field, so do not make things up here. Read the RFC to learn more.

`username (string, default: <NULL>)`
: If no username is set (which means this field does not appear in the configuration file) the server runs in the flag-all-requests-as-error mode. Use your favourite nickname here.

3.4.9 Passthrough Server

3.4.9.1 General description

The program passthrough server provides basic inetd functionality. Basically it can accept connections and pass this connection to the standard input (stdin) and standard output (stdout) handles of programs. Depending on the platform (operating system) the user is able to configure different methods how this can be achieved.

3.4.9.2 Configuration

This server has different types of configuration options specifying its behaviour. Some of them are mandatory and some are optional. The very least to configure is the program to be started when a new connection is made.

`binary (string, no default)`
: This parameter specifies the program to execute when a new connection has been accepted. The parameter is mandatory and must be a fully qualified file name (including path).

`directory (string, no default)`
: This will be the working directory of the executed program. If you omit this parameter the server uses the current directory (the directory is not changed).

`user (string, no default)`
: If you omit this parameter no user or group will be set for the started program. Otherwise you need to specify this information in the format '`user[.group]`'. If the group is omitted the user's primary group will be used.

`argv (string array, no default)`
: This list of character strings is going to be the program's argument list (command line). If the first list item (which is argv[0] and the program's name) is left blank it defaults to the name specified in the `binary` parameter.

`do-fork (boolean, default: true)`
: This flag specifies the method used to pass the connection to the program. If it is true the server uses the Unix'ish `fork` and `exec` method. Otherwise it will pass the data through a unnamed pair of sockets [or two pairs of anonymous pipes].

`single-threaded (boolean, default: true)`
: This parameter applies to servers bound to UDP and ICMP port configurations only. For programs which process all incoming packets and eventually time out,

the program is said to be '`single-threaded`' and should use a true value here. If a program gets a packet and can receive further packets, it is said to be a '`multi-threaded`' program, and should use a false value.

`thread-frequency (integer, default: 40)`
: The optional `thread-frequency` parameter specifies the maximum number of program instances that may be spawned from the server within an interval of 60 seconds.

3.4.10 Mandel Server

3.4.10.1 General description

The distributed Mandelbrot server is an Internet server completely written in Guile with the help of the API provided by the underlying Serveez application. The reader will not see any occurrence of the networking API of Guile.

It implements a protocol called '`dnc`'. '`dnc`' - short for "Distributed Number Cruncher". The Mandelbrot server manages the computation of a graphic visualization of the Mandelbrot set fractal. Each client can connect to the server and ask for something to calculate and is meant to send its result back to the server. Finally the server produces a bitmap in the XPM format and uses a specified viewer application to bring it onto your screen.

3.4.10.2 Configuration

The server can be setup to manage the calculation of the Mandelbrot set at various locations (rectangular region in the complex plane), in a specific pixel resolution and colour depth. Moreover you can define the name of the final output file and the viewer application the output file is displayed with.

`start (string, default: -2.0-1.5i)`
: Specifies the upper left corner of the final bitmap in the complex plane.

`end (string, default: +1.1+1.5i)`
: Specifies the lower right corner of the final bitmap in the complex plane.

`x-res (integer, default: 320)`
: The real part pixel resolution.

`y-res (integer, default: 240)`
: The imaginary part pixel resolution.

`colors (integer, default: 256)`
: Number of maximum colours used in the bitmap. Also determines the maximum iteration depth.

`outfile (string, default: mandel.xpm)`
: When the Mandel server has managed to calculate the whole bitmap it produces an output file in the XPM format. You can specify the name and location of this output file.

`viewer (string, default: xv)`
: Here you can setup your favourite bitmap viewer application. It should be able to parse and display the XPM format.

4 Coserver

4.1 What are coservers

If it is necessary to complete blocking tasks in Serveez you have to use coservers. The actual implementation differs on platforms. On Unices they are implemented as processes communicating with Serveez over pipes. On Win32 Serveez uses threads and shared memory.

4.2 Writing coservers

4.2.1 Making and configuring preparations

First you have to change into the `src/libserveez/coserver/` directory of the Serveez package. Then edit the `Makefile.am` and add your header and implementation file to the `libcoserver_la_SOURCES` variable.

4.2.2 Coserver header file

You have to declare the coserver handle routine here. This callback gets the input buffer argument and delivers the output buffer result. Both of these buffers are supposed to be lines separated by a '`\n`'.

4.2.3 Coserver implementation file

Here you need to `#include "libserveez/coserver/coserver.h"` and implement the coserver handle routine declared in the coserver header file. This can be any blocking system call. On successful completion you can return the result or `NULL` on errors. The input and output buffers are plain strings and can have any format with one exception. Because the coservers communicate via a line protocol with Serveez these buffers must not contain '`\n`' (0x0d).

4.2.4 Make your coserver available in Serveez

For this you have to edit `coserver.h` and `coserver.c` files which are located in the `src/libserveez/coserver/` directory. In the header file you have to define a further `COSERVER_*` id (macro) and set the `MAX_COSERVER_TYPES` define to the appropriate value. Then you should define a further `svz_coserver_*` macro in the same file.

In `coserver.c` you have to implement the `svz_coserver_*` macro. This macro takes three arguments. The first is always specific to your coserver and is used to create the actual request string. Then follows the result callback routine, and an optional argument for this callback. The latter two are simply passed to the `svz_coserver_send_request` routine. This routine takes four arguments where the first is the previously defined `COSERVER_*` id and the second is the input buffer for the coserver handle routine without the trailing '`\n`'.

Then you need to add your coserver to the `svz_coservertypes` array specifying the `COSERVER_*` id, the coserver description, the coserver handle routine discussed above, the number of coserver instances to be created and an optional initialization routine.

4.3 Existing coservers

4.3.1 Identification (Ident) coserver

The Identification protocol is briefly documented in RFC1413. It provides a means to determine the identity of a user of a particular TCP connection. Given a TCP port number pair, it returns a character string which identifies the owner of that connection on the server's (that is the client's) system.

This is a connection based application on TCP. A server listens for TCP connections on TCP port 113 (decimal). Once a connection is established, the server reads a line of data which specifies the connection of interest. If it exists, the system dependent user identifier of the connection of interest is sent as the reply. The server may then either shut down the connection or it may continue to read/respond to more queries.

The Ident coserver is a client to this kind of service. For every established network connection you can use this service by calling the appropriate macro from `coserver.h`. But you could also use the Ident coserver as is without this macro. The messages from Serveez to this coserver are formatted this way:

```
Format:
RemoteAddressInDottedDecimals ":" RemotePort ":" LocalPort

Macro:
svz_coserver_ident (sock, MyIdentCallback, sock->id, sock->version);
```

In this context `sock` is of type `svz_socket_t` and `MyIdentCallback` is something like the following example. Both of the last two (optional) arguments identify a valid socket structure and `user` can be `NULL` if there is no ident daemon running on the foreign machine. The last two argument within the above macro will be the last two arguments in the callback below. Thus you will know what kind of data the invocation of the callback is related to.

```
Callback:
int
MyIdentCallback (char *user, int id, int version)
{
  printf ("Identified user: %s\n", user);
  return 0;
}
```

4.3.2 Domain Name Server (DNS) coserver

The DNS coserver is using `gethostbyname` to translate a given hostname to the associated IP address. The format of the coserver input line and the macro from `coserver.h` is shown below. The IRC server is currently using this coserver for resolving its '`?-Lines`'. See Section 3.4 [Existing servers], page 36, for more information. In the example below `realhost` is something like '`www.lkcc.org`'.

```
Format:
RemoteHostname

Macro:
svz_coserver_dns (realhost, irc_connect_server, ircserver, NULL);

Callback:
```

```
int
irc_connect_server (char *ip, irc_server_t *server)
{
  printf ("The ip address is: %s\n", ip);
  return 0;
}
```

4.3.3 Reverse Domain Name Server (reverse DNS) coserver

As easily guessed from the name this coserver is just doing the reverse as the DNS coserver. It translates a given IP address into a hostname using `gethostbyaddr`. In the macro the ip address is given as an `unsigned long` in host byte order. The Reverse DNS coserver itself takes something like '`192.168.2.1`'.

```
Format:
RemoteAddressInDottedDecimals

Macro:
svz_coserver_reverse (addr, MyReverseCallback, sock->id, sock->version);

Callback:
int
MyReverseCallback (char *host, int id, int version)
{
  printf ("Hostname is: %s\n", host);
  return 0;
}
```

5 Embedding

This chapter documents how to embed Serveez into C programs and describes all parts of the API it provides.

The Serveez core library provides all of the functionality necessary in order to write Internet protocol servers (currently TCP, UDP, ICMP and RAW sockets), pipe servers (connection-oriented via a pair of named pipes) and coservers in a portable way. All of the included servers are based upon this library, which encapsulates the native network and file system programming interface of different Unices and Windows systems.

The following sections will give the reader an overview about how to use its core library.

5.1 Embedding Serveez

This chapter deals with embedding the Serveez core library into standalone C/C++ applications and using it in order to write additional servers.

5.1.1 Compiling and linking

When you have installed a version of Serveez passing the `configure` script the `$prefix` argument, e.g., '`./configure --prefix=/usr/local`', you will find the `libserveez` library in `/usr/local/lib` and the include headers in `/usr/local/include`. If you want to compile a C program using the Serveez API and link against the Serveez core library `libserveez`, which is `libserveez.so` for Unices and `libserveez.dll` for Windows systems, you need to tell the compiler and linker where to find the headers and libraries.

Most C compilers you can use will understand the following command line options for this purpose. The '`-I`' argument specifies the directory of additional include headers, the '`-L`' argument the direcory to additional libraries and the '`-l`' argument the library itself to link against.

```
$ cc test.c -I/usr/local/include -o test -L/usr/local/lib -lserveez
```

In order to obtain the correct compiler and linker flag you can also run the '`serveez-config`' script which gets installed with the Serveez package. The '`serveez-config`' script can be invoked with the following set of option.

`-h, --help`
: Displays the usage information.

`-v, --version`
: Displays installed Serveez version.

`-l, --ldflags`
: Prints the linker flags (libraries to link with including directory information).

`-c, --cflags`
: Prints the compiler flags to compile with.

5.1.2 A simple example

The following small example shows how to use the Serveez core library to print the list of known network interface. As you will notice there are three major steps to do: Include the library header with `#include <libserveez.h>`, initialize the library via `svz_boot` and

finalize it via `svz_halt`. In between these calls you can use all of the API functions, variables and macros described in Section 5.2 [Embedding API], page 55.

```
#include <stdio.h>
#include <stdlib.h>
#include <libserveez.h>

static int
display_ifc (const svz_interface_t *ifc,
             void *closure)
{
  char *addr = svz_inet_ntoa (ifc->ipaddr);

  if (ifc->description)
    /* interface with description */
    printf ("%40s: %s\n",
            ifc->description, addr);
  else
    /* interface with interface # only */
    printf ("%31s%09lu: %s\n",
            "interface # ", ifc->index, addr);
  return 0;
}

int
main (int argc, char **argv)
{
  /* Library initialization.  */
  svz_boot ("example");

  /* Display a list of interfaces.  */
  printf ("local interfaces:\n");
  svz_foreach_interface (display_ifc, NULL);

  /* Library finalization.  */
  svz_halt ();

  return EXIT_SUCCESS;
}
```

5.2 Embedding API

In this chapter the reader will find a short description of each function, global variable and macro provided by the Serveez core library. The API can either be used to implement a new server or coserver module for use with Serveez or for supporting network and server functionality within your own applications without caring about the details and system programming.

Most of the Serveez core library interface functionality should be prefixed with `svz_`. Small symbols will refer to functions and variables in most cases and big letter symbols refer to macros.

[FIXME: The subsections are named inconsistently because originally, the API reference was a separate document; on merge, weirdness like "Port config funcs" was necessary to avoid conflict with the other "Port configuration" node. —ttn]

5.2.1 Library features

The `configure` script used to build libserveez takes many options (see Section 1.1 [Build and install], page 1). Some of these are encapsulated by `svz_library_features`.

`const char * const *` [Function]
`svz_library_features` (*size_t *count*)
Return a list (length saved to *count*) of strings representing the features compiled into libserveez.

Here is a table describing the features in detail:

`debug` Present when '`--enable-debug`'.

`heap-counters`
Present when '`--enable-heap-count`'.

`interface-list`
Present when '`--enable-iflist`'.

`poll` Present when '`--enable-poll`' and you have poll(2).

`sendfile` Present when '`--enable-sendfile`' and you have sendfile(2) or some workalike (e.g., `TransmitFile`).

`log-mutex`
Present when `svz_log` uses a mutex around its internal stdio operations, implying that you have some kind of thread capability (perhaps in a separate library). If your system has `fwrite_unlocked`, the configure script assumes that `fwrite` et al already operate in a locked fashion, and disables this.

`flood-protection`
Present when '`--enable-flood`'.

`core` The networking core. This is always present.

5.2.2 Memory management

The core library of Serveez is able to keep track of the memory an application or part of a program consumes, and also controls itself in the same manner. When you are using this memory allocator interface you can determine and afterwards remove memory leaks. This is a very important feature as servers are by nature long-lived programs.

The three allocator function pointers for `malloc`, `realloc` and `free` make it possible to instruct Serveez to use different kinds of memory, which might be necessary if you want the library to work with shared memory arenas or any other underlying memory API.

`void` [Function]
`svz_set_mm_funcs` (*svz_malloc_func_t* **`cus_malloc`**,
svz_realloc_func_t **`cus_realloc`**, *svz_free_func_t* **`cus_free`**)
Set the internal memory management functions to *cus_malloc*, *cus_realloc* and *cus_free*, respectively. The default internal values are `malloc`, `realloc` and `free`.

`void *` [Function]
`svz_malloc` (*size_t* **`size`**)
Allocate *size* bytes of memory and return a pointer to this block.

`void *` [Function]
`svz_calloc` (*size_t* **`size`**)
Allocate *size* bytes of memory and return a pointer to this block. The memory is cleared (filled with zeros).

`void *` [Function]
`svz_realloc` (*void* ***`ptr`**, *size_t* **`size`**)
Change the size of a block of memory at *ptr*, previously returned by `svz_malloc`, to *size* bytes. If *ptr* is `NULL`, allocate a new block.

`void` [Function]
`svz_free` (*void* ***`ptr`**)
Free a block of memory at *ptr*, previously returned by `svz_malloc` or `svz_realloc`. If *ptr* is `NULL`, do nothing.

`char *` [Function]
`svz_strdup` (*const char* ***`src`**)
Duplicate the given string *src* if it is not `NULL` and has non-zero length. Return the new string.

`void` [Function]
`svz_get_curalloc` (*size_t* ***`to`**)
Write values to `to[0]` and `to[1]` representing the number of currently allocated bytes and blocks, respectively. If Serveez was not configured with '`--enable-debug`', the values are always 0.

5.2.3 Data structures

Since most servers need to store information about its clients or want to keep track of data during runtime, we include a pair of of useful data structures. The actual aim was to provide higher level data structures which the C programming language does not support. Some of the included servers which come with Serveez make extensive use of them.

5.2.3.1 Array

The array data structure is a simple array implementation. Each array has a size and capacity. The array indices range from zero to the array's size minus one. You can put any kind of data into this array which fits into the size of a pointer. The array grows automatically if necessary.

`svz_array_t *` [Function]
`svz_array_create` (*size_t* `capacity`, *svz_free_func_t* `destroy`)
Create a new array with the initial capacity *capacity* and return a pointer to it. If *capacity* is zero it defaults to some value. If *destroy* is non-`NULL`, `svz_array_destroy` calls that function (typically used to free dynamically allocated memory). For example, if the array contains data allocated by `svz_malloc`, *destroy* should be specified as `svz_free`. If the array contains data which should not be released, *destroy* should be `NULL`.

`void` [Function]
`svz_array_destroy` (*svz_array_t* *`array`)
Completely destroy the array *array*. The *array* handle is invalid afterwards. The routine runs the *destroy* callback for each element of the array.

`void *` [Function]
`svz_array_get` (*svz_array_t* *`array`, *size_t* `index`)
Return the array element at the position *index* of the array *array* if the index is within the array range. Return `NULL` if not.

`void *` [Function]
`svz_array_set` (*svz_array_t* *`array`, *size_t* `index`, *void* *`value`)
Replace the array element at the position *index* of the array *array* with the value *value* and return the previous value at this index. Return `NULL` and do nothing if *array* is `NULL` or the *index* is out of the array range.

`void` [Function]
`svz_array_add` (*svz_array_t* *`array`, *void* *`value`)
Append the value *value* at the end of the array *array*. Do nothing if *array* is `NULL`.

`void *` [Function]
`svz_array_del` (*svz_array_t* *`array`, *size_t* `index`)
Remove the array element at the position *index* of the array *array*. Return its previous value or `NULL` if the index is out of the array's range.

`size_t` [Function]
`svz_array_size` (*svz_array_t* *`array`)
Return the current size of *array*.

`svz_array_foreach` (*array*, *value*, *i*) [Macro]
Expand into a `for`-statement header, for iterating over *array*. On each cycle, *value* is assigned to successive elements of *array*, and *i* the element's position.

5.2.3.2 Hashtable

A hashtable associates keys of arbitrary size and content with values. This data structure is also called associative array sometimes because you use keys in order to access values instead of numbers. You cannot store two values associated with the same key. The values can have any simple C types like integers or pointers.

`svz_hash_t *` [Function]
`svz_hash_create` (*size_t* **`size`**, *svz_free_func_t* **`destroy`**)

Create a new hash table with an initial capacity *size*. Return a non-zero pointer to the newly created hash. The size is calculated down to a binary value. The *destroy* callback specifies an element destruction callback for use by `svz_hash_clear` and `svz_hash_destroy` for each value. If no such operation should be performed the argument must be `NULL`.

`svz_hash_t *` [Function]
`svz_hash_configure` (*svz_hash_t* ***`hash`**, *size_t* (* *keylen*) (*const char* *) , *unsigned long* (* *code*) (*const char* *) , *int* (* *equals*) (*const char* *, *const char* *)

Set the internal *keylen*, *code* and and *equals* functions for hash table *hash*. Return *hash*.

keylen takes `const char *data` and returns `size_t`, the number of bytes in *data* representing the key.

code takes `const char *data` and returns `unsigned long`.

equals takes `const char *data1, const char *data2` and returns `int`, which should be non-zero if equal.

As a special case, a `NULL` value means don't set that function, leaving it to its default value.

`void` [Function]
`svz_hash_destroy` (*svz_hash_t* ***`hash`**)

Destroy the existing hash table *hash*, `svz_free`ing all keys within the hash, the hash table and the hash itself. If a non-`NULL` element destruction callback was specified to `svz_hash_create`, that function is called on each value.

`void *` [Function]
`svz_hash_delete` (*svz_hash_t* ***`hash`**, *const char* ***`key`**)

Delete an existing entry accessed via a *key* from the hash table *hash*. Return `NULL` if there is no such key, otherwise the previous value.

`void *` [Function]
`svz_hash_put` (*svz_hash_t* ***`hash`**, *const char* ***`key`**, *void* ***`value`**)

Add a new element consisting of *key* and *value* to *hash*. When *key* already exists, replace and return the old value. **Note**: This is sometimes the source of memory leaks.

`void *` [Function]
`svz_hash_get` (*const svz_hash_t* ***`hash`**, *const char* ***`key`**)

Return the value associated with *key* in the hash table *hash*, or `NULL` if there is no such key.

`void` [Function]
`svz_hash_foreach` (*svz_hash_do_t* ***`func`**, *svz_hash_t* ***`hash`**, *void* ***`closure`**)

Iterate *func* over each key/value pair in *hash*. *func* is called with three `void *` args: the key, the value and the opaque (to `svz_hash_foreach`) *closure*.

`size_t` [Function]
`svz_hash_size` (*const svz_hash_t* ***hash***)
Return the number of keys in the hash table *hash*. If *hash* is `NULL`, return zero.

`char *` [Function]
`svz_hash_contains` (*const svz_hash_t* ***hash***, *void* ***value***)
Return the key associated with *value* in the hash table *hash*, or `NULL` if there is no such value.

`int` [Function]
`svz_hash_exists` (*const svz_hash_t* ***hash***, *char* ***key***)
Return non-zero if `key` is stored within the hash table `hash`, otherwise zero. This function is useful when you cannot tell whether the return value of `svz_hash_get` (`== NULL`) indicates a real value in the hash or a non-existing hash key.

5.2.4 svz_address_t

A network address comprises a *family*, such as `AF_INET` (also known as IPv4), and its *bits* in network byte order, such as the bytes 127, 0, 0 and 1 (also known as the *IPv4 loopback address*). Many libserveez functions take `svz_address_t *`.

Please note: Although `svz_address_t` supports[1] IPv6, the rest of libserveez it does not (yet). This means you can freely create and manipulate address objects with the functions described in this section, but any attempt to pass to the rest of libserveez an address with a *family* other than `AF_INET` will immediately abort the process. When full IPv6 support is in place, this blurb will be deleted and the list returned by `svz_library_features` will include an appropriate indicator (see Section 5.2.1 [Library features], page 56).

`svz_address_t *` [Function]
`svz_address_make` (*int* ***family***, *const void* ***bits***)
Return an address object to hold an address in *family*, represented by *bits*. *family* must be one of:

`AF_INET` An IPv4 address; *bits* is `in_addr_t *`.

`AF_INET6` (if supported by your system) An IPv6 address; *bits* is `struct in6_addr *`.

The *bits* are expected in network byte order. If there are problems, return `NULL`.

`int` [Function]
`svz_address_family` (*const svz_address_t* ***addr***)
Return the address family of *addr*.

`int` [Function]
`svz_address_to` (*void* ***dest***, *const svz_address_t* ***addr***)
Copy the address bits out of *addr* to *dest*. Return 0 on success, -1 if either *addr* or *dest* is `NULL`, or the *addr* family is `AF_UNSPEC`.

[1] that is, if your system supports it

`int` [Function]
`svz_address_same` (*const svz_address_t* *`a`, *const svz_address_t* *`b`)
Return 1 if *a* and *b* represent the same address (identical family and bits), otherwise 0.

`const char *` [Function]
`svz_pp_address` (*char* *`buf`, *size_t* `size`, *const svz_address_t* *`addr`)
Format an external representation of *addr* into *buf*, of *size* bytes. The format depends on the family of *addr*. For IPv4, this is numbers-and-dots. For IPv6, it is "the most appropriate IPv6 network address format for *addr*", according to the manpage of `inet_ntop`, the function that actually does the work.

If *buf* or *addr* is `NULL`, or *size* is not big enough, return `NULL`. Otherwise, return *buf*.

`const char *` [Function]
`svz_pp_addr_port` (*char* *`buf`, *size_t* `size`, *const svz_address_t* *`addr`, *in_port_t* `port`)
Format an external representation of *addr* and *port* (in network byte order) into *buf*, of *size* bytes. The address *xrep* (external representation) is done by `svz_pp_address`, q.v. The rest of the formatting depends on the *addr* family.

Family	**Formatting**
`AF_INET` (IPv4)	*xrep*`:`*port*
`AF_INET6` (IPv6)	`[`*xrep*`]:`*port*

If *buf* or *addr* is `NULL`, or *size* is not big enough, return `NULL`. Otherwise, return *buf*.

`svz_address_t *` [Function]
`svz_address_copy` (*const svz_address_t* *`addr`)
Return a copy of *addr*.

`SVZ_SET_ADDR` (*place*, *family*, *bits*) [Macro]
Expand to a series of commands. First, if *place* is non-`NULL`, then `svz_free` it. Next, assign to *place* a new address object made by calling `svz_address_make` with *family* and *bits*.

`SVZ_PP_ADDR` (*buf*, *addr*) [Macro]
Expand to a call to `svz_pp_address`, passing it *buf* and `sizeof` *buf*, in addition to *addr*.

`SVZ_PP_ADDR_PORT` (*buf*, *addr*, *port*) [Macro]
Expand to a call to `svz_pp_addr_port`, passing it *buf* and `sizeof` *buf*, in addition to *addr* and *port*.

5.2.5 Utility functions

Within this section you will find some miscellaneous functionality and left overs of the C API.

`void` [Function]
`svz_log` (*int* `level`, *const char* *`format`, ...)
Print a message to the log system. *level* specifies the prefix.

`void` [Function]
`svz_log_setfile` (*FILE* **file*)
Set the file stream *file* to the log file all messages are printed to. Can also be `stdout` or `stderr`.

`int` [Function]
`svz_hexdump` (*FILE* **out*, *char* **action*, *int from*, *char* **buffer*, *int len*, *int max*)
Dump *buffer* with the length *len* to the file stream *out*. Display description *action* along with origin and size info first, followed by the hexadecimal text representation. Stop output at either *max* or *len* (if *max* is zero) bytes. *from* is a numerical identifier of the buffers creator.

`char *` [Function]
`svz_itoa` (*unsigned int i*)
Convert an unsigned integer to its decimal string representation, returning a pointer to an internal buffer. (You should copy the result.)

`unsigned int` [Function]
`svz_atoi` (*char* **str*)
Convert string *str* in decimal format to an unsigned integer. Stop conversion on any invalid characters.

`char *` [Function]
`svz_getcwd` (*void*)
Return the current working directory in a newly allocated string. (You should `svz_free` it when done.)

`int` [Function]
`svz_openfiles` (*int max_sockets*)
Check for the current and maximum limit of open files of the current process and try to set the limit to *max_sockets*.

`char *` [Function]
`svz_time` (*long t*)
Transform the given binary data *t* (UTC time) to an ASCII time text representation without any trailing characters.

`char *` [Function]
`svz_tolower` (*char* **str*)
Convert the given string *str* to lower case text representation.

`char *` [Function]
`svz_sys_version` (*void*)
Return a statically-allocated string describing some operating system version details.

`int` [Function]
`svz_socket_unavailable_error_p` (*void*)
Return 1 if there was a "socket unavailable" error recently, 0 otherwise. This checks `svz_errno` against `WSAEWOULDBLOCK` (woe32) or `EAGAIN` (Unix).

`const char *` [Function]
`svz_sys_strerror` (*void*)
Return a string describing the most recent system error.

The next two functions log (with `SVZ_LOG_ERROR`) the current *system error* or *network error*, forming the prefix of the message using *fmt* and *args*. This formatted prefix cannot exceed 255 bytes. The rest of the message comprises: colon, space, error description, newline.

`void` [Function]
`svz_log_sys_error` (*char const *`fmt`, ...*)
Log the current *system error*.

`void` [Function]
`svz_log_net_error` (*char const *`fmt`, ...*)
Log the current *network error*.

`int` [Function]
`svz_mingw_at_least_nt4_p` (*void*)
Return 1 if running MinGW (Windows) NT4x or later, otherwise 0.

5.2.6 Networking and other low level functions

This chapter deals with the basic networking and file systems functions. It encapsulates systems calls in a portable manner. These functions should behave identically on Windows and Unices.

`char *` [Function]
`svz_inet_ntoa` (*in_addr_t `ip`*)
Convert *ip*, an address in network byte order, to its dotted decimal representation, returning a pointer to a statically allocated buffer. (You should copy the result.)

`int` [Function]
`svz_inet_aton` (*char *`str`, struct sockaddr_in *`addr`*)
Convert the Internet host address *str* from the standard numbers-and-dots notation into binary data and store it in the structure that *addr* points to. Return zero if the address is valid, nonzero otherwise. As a special case, if *str* is '`*`' (asterisk), store `INADDR_ANY` in *addr*.

`int` [Function]
`svz_closesocket` (*svz_t_socket `sockfd`*)
Close the socket *sock*. Return 0 if successful, -1 otherwise.

`int` [Function]
`svz_fd_cloexec` (*int `fd`*)
Set the close-on-exec flag of the given file descriptor *fd* and return zero on success. Otherwise return non-zero.

`int` [Function]
`svz_tcp_cork` (*svz_t_socket* `fd`, *int* `set`)
Enable or disable the `TCP_CORK` socket option of the socket *fd*. This is useful for performance reasons when using `sendfile` with any prepending or trailing data not inside the file to transmit. Return zero on success, otherwise non-zero.

`int` [Function]
`svz_tcp_nodelay` (*svz_t_socket* `fd`, *int* `set`, *int* *`old`)
Enable or disable the `TCP_NODELAY` setting for the socket *fd* depending on the flag *set*, effectively enabling or disabling the Nagle algorithm. This means that packets are always sent as soon as possible and no unnecessary delays are introduced. If *old* is not `NULL`, save the old setting there. Return zero on success, otherwise non-zero.

`int` [Function]
`svz_sendfile` (*int* `out_fd`, *int* `in_fd`, *off_t* *`offset`, *size_t* `count`)
Transmit data between one file descriptor and another where *in_fd* is the source and *out_fd* the destination. The *offset* argument is a pointer to a variable holding the input file pointer position from which reading starts. On return, the *offset* variable will be set to the offset of the byte following the last byte that was read. *count* is the number of bytes to copy. Return the number of bytes actually read/written or -1 on errors.

`int` [Function]
`svz_open` (*const char* *`file`, *int* `flags`, *mode_t* `mode`)
Open the filename *file* and convert it into a file handle. The given *flags* specify the access mode and the *mode* argument the permissions if the `O_CREAT` flag is set.

`int` [Function]
`svz_close` (*int* `fd`)
Close the given file handle *fd*. Return -1 on errors.

`int` [Function]
`svz_fstat` (*int* `fd`, *struct stat* *`buf`)
Return information about the specified file associated with the file descriptor *fd* returned by `svz_open`. Store available information in the stat buffer *buf*.

`FILE *` [Function]
`svz_fopen` (*const char* *`file`, *const char* *`mode`)
Open the file whose name is the string pointed to by *file* and associate a stream with it.

`int` [Function]
`svz_fclose` (*FILE* *`f`)
Dissociate the named stream *f* from its underlying file.

5.2.7 Client connections

Serveez tries to handle all kinds of Internet protocols like TCP (connection oriented), UDP, ICMP and RAW (packet oriented) and communication across named pipes (also connection oriented) in the same way. Therefore it uses a structure called `svz_socket_t` which is the abstraction of any kind of communication endpoint (can be client or server or both together).

5.2.7.1 TCP sockets

TCP sockets provide a reliable, stream oriented, full duplex connection between two sockets on top of the Internet Protocol (IP). TCP guarantees that the data arrives in order and retransmits lost packets. It generates and checks a per packet checksum to catch transmission errors. TCP does not preserve record boundaries.

`svz_socket_t *` [Function]
`svz_tcp_connect` (*svz_address_t* ***host**, *in_port_t* **port**)
Create a TCP connection to host *host* and set the socket descriptor in structure *sock* to the resulting socket. Return `NULL` on errors.

`int` [Function]
`svz_tcp_read_socket` (*svz_socket_t* ***sock**)
Read all data from *sock* and call the `check_request` function for the socket, if set. Return -1 if the socket has died, zero otherwise.

This is the default function for reading from *sock*.

`int` [Function]
`svz_tcp_send_oob` (*svz_socket_t* ***sock**)
If the underlying operating system supports urgent data (out-of-band) in TCP streams, try to send the byte in `sock->oob` through the socket structure *sock* as out-of-band data. Return zero on success and -1 otherwise (also if urgent data is not supported).

5.2.7.2 Pipe connections

The pipe implementation supports both named and anonymous pipes. Pipe servers are implemented as listeners on a file system FIFO on Unices or "Named Pipes" on Windows (can be shared over a Windows network).

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

`svz_socket_t *` [Function]
`svz_pipe_create` (*svz_t_handle* **recv_fd**, *svz_t_handle* **send_fd**)
Create a socket structure containing both the pipe descriptors *recv_fd* and *send_fd*. Return `NULL` on errors.

`int` [Function]
`svz_pipe_create_pair` (*svz_t_handle* **pipe_desc**[*2*])
Create a (non blocking) pair of pipes. This differs in Win32 and Unices. Return a non-zero value on errors.

`svz_socket_t *` [Function]
`svz_pipe_connect` (*svz_pipe_t* ***recv**, *svz_pipe_t* ***send**)
Create a pipe connection socket structure to the pair of named pipes *recv* and *send*. Return `NULL` on errors.

`int` [Function]
`svz_invalid_handle_p` (*svz_t_handle* **handle**)
Return 1 if *handle* is invalid, otherwise 0.

`void` [Function]
`svz_invalidate_handle` (*svz_t_handle* ***href**)
Invalidate the handle pointed at by *href*.

`int` [Function]
`svz_closehandle` (*svz_t_handle* **handle**)
Close *handle*. Return 0 if successful, -1 otherwise.

5.2.7.3 UDP sockets

The UDP sockets implement a connectionless, unreliable datagram packet service. Packets may be reordered or duplicated before they arrive. UDP generates and checks checksums to catch transmission errors.

`svz_socket_t *` [Function]
`svz_udp_connect` (*svz_address_t* ***host**, *in_port_t* **port**)
Create a UDP connection to *host* at *port* and set the socket descriptor in structure *sock* to the resulting socket. Return a `NULL` value on errors.

This function can be used for port bouncing. If you assign the `handle_request` callback to something server specific and the *cfg* field of the server's configuration to the returned socket structure, this socket is able to handle a dedicated UDP connection to some other UDP server.

`int` [Function]
`svz_udp_write` (*svz_socket_t* ***sock**, *char* ***buf**, *int* **length**)
Write *buf* into the send queue of the UDP socket *sock*. If *length* argument supersedes the maximum length for UDP messages it is split into smaller packets.

5.2.7.4 ICMP sockets

The ICMP socket implementation is currently used in the tunnel server which comes with the Serveez package. It implements a user protocol receiving and sending ICMP packets by opening a raw socket with the protocol `IPPROTO_ICMP`.

The types of ICMP packets passed to the socket can be filtered using the `ICMP_FILTER` socket option (or by software as done here). ICMP packets are always processed by the kernel too, even when passed to a user socket.

`svz_socket_t *` [Function]
`svz_icmp_connect` (*svz_address_t* ***host**, *in_port_t* **port**, *uint8_t* **type**)
Create an ICMP socket for receiving and sending. Return `NULL` on errors, otherwise an enqueued socket structure.

`int` [Function]
`svz_icmp_send_control` (*svz_socket_t* ***sock**, *uint8_t* **type**)
"If you are calling this function we will send an empty ICMP packet signaling that this connection is going down soon." [ttn sez: huh?]

`int` [Function]
`svz_icmp_write` (*svz_socket_t* ***sock**, *char* ***buf**, *int* **length**)
Send *buf* with length *length* via this ICMP socket *sock*. If *length* supersedes the maximum ICMP message size the buffer is split into smaller packets.

5.2.7.5 Raw sockets

A raw socket receives or sends the raw datagram not including link-level headers. It is currently used by the ICMP socket implementation of the core library. The IPv4 layer generates an IP header when sending a packet unless the `IP_HDRINCL` socket option is enabled on the socket. When it is enabled, the packet must contain an IP header. For receiving the IP header is always included in the packet.

Only processes with an effective userid of zero (Administrator or root) or the `CAP_NET_RAW` capability are allowed to open raw sockets. All packets or errors matching the protocol number specified for the raw socket are passed to this socket. A protocol of `IPPROTO_RAW` implies enabled `IP_HDRINCL` and receives all IP protocols. Sending is not allowed.

[FIXME: All funcs internalized! Write something else here!]

5.2.7.6 Passthrough connections

The functions described in this section allow you to pass through client connections to the standard input (stdin) and standard output (stdout) of external programs. Some of the routines deal with the management of program environments. Basically, there are two methods for passing through a duplex connection: the Unix'ish `fork` and `exec` method and the shuffle method where the main process keeps control over the communication on the original duplex connection and passes this data through two pairs of pipes, or yet another socket connection, to the child process. All of the three method are implemented calling them `SVZ_PROCESS_FORK`, `SVZ_PROCESS_SHUFFLE_PIPE` and `SVZ_PROCESS_SHUFFLE_SOCK`.

`int` [Function]
`svz_sock_process` (*svz_socket_t* ***sock**, *char* ***bin**, *char* ***dir**, *char* ****argv**, *svz_envblock_t* ***envp**, *int* **forkp**, *char* ***user**)
Start a new program *bin*, a fully qualified executable filename, passing the socket or pipe descriptor(s) in the socket structure *sock* to its stdin and stdout.

If *dir* is non-`NULL`, it specifies the working directory of the new process.

The program arguments and the environment of the new process are taken from *argv* and *envp*. Normally `argv[0]` should be set to the program's name. If `NULL`, it defaults to *bin*.

The *forkp* argument is a flag that controls the passthrough method. If non-zero, pipe descriptors or the socket descriptor are passed to the child process directly through `fork` and `exec`. Otherwise, socket transactions are passed via a pair or pipes or sockets (depending on whether or not the system provides `socketpair`).

You can pass the user and group identifications in the format '`user[.group]`' (group is optional), as `SVZ_PROCESS_NONE` or `SVZ_PROCESS_OWNER` in the *user* argument. This specifies the permissions of the new child process. If `SVZ_PROCESS_OWNER` is passed the permissions are set to the executable file *bin* owner; `SVZ_PROCESS_NONE` does not change user or group.

Return the new process id on success, -1 on failure.

Please note: On M$-Windows platforms it is not possible to pass a socket connection to stdin/stdout of a child process. That is why this function creates an inheritable version of the socket and puts the socket handle number into the environment variables `SEND_HANDLE` and `RECV_HANDLE`. A spawned child process can use these handles as if they were self-created. After calling `WSAStartup` the child process can `send` and `recv` as usual.

Relatedly, Windoze does not use `SIGCHLD` to inform the parent when a child dies, so for that platform, you should use the next function (which is not otherwise available):

`int` [Function]
`svz_mingw_child_dead_p` (*char *prefix*, *svz_t_handle *pid*)
Check child pointed at by *pid* by waiting a bit. If it is dead, close and invalidate its handle, and return 1. Otherwise, return 0. *prefix* is for error messages; it should be either the empty string, or a string ending in colon and space.

On non-Windoze, this is the function you want to use:

`int` [Function]
`svz_most_recent_dead_child_p` (*svz_t_handle pid*)
Return 1 if a child process *pid* died recently, updating other internal state by side effect. Otherwise, return 0.

`void` [Function]
`svz_envblock_setup` (*void*)
Set up internal tables for environment block wrangling.

This function must be called once after `svz_boot` so that subsequent functions (like `svz_envblock_default`) can work correctly.

`svz_envblock_t *` [Function]
`svz_envblock_create` (*void*)
Create and return a fresh environment block, useful for passing to `svz_envblock_default` and `svz_envblock_add`. Its size is initially set to zero.

`int` [Function]
`svz_envblock_default` (*svz_envblock_t *env*)
Fill environment block *env* with the environment variables from the current process, replacing its current contents (if any).

`int` [Function]
`svz_envblock_add` (*svz_envblock_t *env*, *char *format*, ...)
Insert a new environment variable into environment block *env*. The *format* argument is a `printf`-style format string describing how to format the optional arguments. You specify environment variables in the '`VAR=VALUE`' format.

`void` [Function]
`svz_envblock_destroy` (*svz_envblock_t *env*)
Destroy environment block *env* completely. Afterwards, *env* is invalid and should therefore not be further referenced.

`void *` [Function]
`svz_envblock_get` (*svz_envblock_t* ***env***)

Convert environment block *env* into something which can be passed to `execve` (Unix) or `CreateProcess` (Windows). Additionally, under Windows, sort the environment block.

(Unfortunately the layout of environment blocks in Unices and Windows differ. On Unices you have a NULL terminated array of character strings (i.e., `char **`) and on Windows systems you have a simple character string containing the environment variables in the format '`VAR=VALUE`' each separated by a zero byte (i.e., `char *`). The end of the list is indicated by a further zero byte.)

5.2.8 Socket management

The functions described in this section deal with the operations on C structures called `svz_socket_t`. See the description of each function for details on which kind of socket it can handle and what they are for.

`int` [Function]
`svz_sock_nconnections` (*void*)

Return the number of currently connected sockets.

`int` [Function]
`svz_sock_write` (*svz_socket_t* ***sock***, *char* ***buf***, *int* ***len***)

Write *len* bytes from the memory location pointed to by *buf* to the output buffer of the socket *sock*. Also try to flush the buffer to the socket of *sock* if possible. Return a non-zero value on error, which normally means a buffer overflow.

`int` [Function]
`svz_sock_printf` (*svz_socket_t* ***sock***, *const char* ***fmt***, ...)

Print a formatted string on the socket *sock*. *fmt* is the `printf`-style format string, which describes how to format the optional arguments.

`int` [Function]
`svz_sock_resize_buffers` (*svz_socket_t* ***sock***, *int* ***send_buf_size***, *int* ***recv_buf_size***)

Resize the send and receive buffers for the socket *sock*. *send_buf_size* is the new size for the send buffer, *recv_buf_size* for the receive buffer. Note that data may be lost when the buffers shrink. For a new buffer size of 0 the buffer is freed and the pointer set to NULL.

`int` [Function]
`svz_sock_check_request` (*svz_socket_t* ***sock***)

Check for the kind of packet delimiter within *sock* and and assign one of the default `check_request` routines (one or more byte delimiters or a fixed size).

Afterwards this function will never ever be called again because the callback gets overwritten here.

`void` [Function]
`svz_sock_reduce_recv` (*svz_socket_t* ***sock***, *int* ***len***)

Shorten the receive buffer of *sock* by *len* bytes.

`void` [Function]
`svz_sock_reduce_send` (*svz_socket_t* ***sock**, *int* **len**)
Reduce the send buffer of *sock* by *len* bytes.

Because libserveez manages the creation and destruction of `svz_socket_t` objects internally, the following API element is useful for syncronizing client-code references to those objects with those objects.

`void` [Function]
`svz_sock_prefree` (*int* **addsub**, *svz_sock_prefree_fn* **fn**)
Register (if *addsub* is non-zero), or unregister (otherwise) the function *fn* to be called immediately prior to a `svz_socket_t` being freed. *fn* is called with one arg `sock`, and should not return anything. In other words:

```
typedef void (svz_sock_prefree_fn) (const svz_socket_t *);
```

Note the `const`!

5.2.9 Coserver functions

This section describes the internal coserver interface of Serveez. Coservers are helper processes meant to perform blocking tasks. This is necessary because Serveez itself is single threaded. Each coserver is connected via a pair of pipes to the main thread of Serveez communicating over a simple text line protocol. Each request/response is separated by a newline character.

`int` [Function]
`svz_foreach_coserver` (*svz_coserver_do_t* ***func**, *void* ***closure**)
Call *func* for each coserver, passing additionally the second arg *closure*. If *func* returns a negative value, return immediately with that value (breaking out of the loop), otherwise, return 0.

`void` [Function]
`svz_coserver_check` (*void*)
Under woe32 check if there was any response from an active coserver. Moreover keep the coserver threads/processes alive. If one of the coservers dies due to buffer overrun or might be overloaded, start a new one.

Call this function whenever there is time, e.g., within the timeout of the `select` system call.

`void` [Function]
`svz_coserver_destroy` (*int* **type**)
Destroy specific coservers with the type *type*. All instances of this coserver type will be stopped.

`svz_coserver_t *` [Function]
`svz_coserver_create` (*int* **type**)
Create and return a single coserver with the given type *type*.

`const char *` [Function]
`svz_coserver_type_name` (*const svz_coserver_t* ***coserver**)
Return the type name of *coserver*.

`void` [Function]
`svz_coserver_rdns_invoke` (*svz_address_t* ***addr**, *svz_coserver_handle_result_t* **cb**, *void* ***closure**)

Enqueue a request for the reverse DNS coserver to resolve address *addr*, arranging for callback *cb* to be called with two args: the hostname (a string) and the opaque data *closure*.

`void` [Function]
`svz_coserver_dns_invoke` (*char* ***host**, *svz_coserver_handle_result_t* **cb**, *void* ***closure**)

Enqueue a request for the DNS coserver to resolve *host*, arranging for callback *cb* to be called with two args: the ip address in dots-and-numbers notation and the opaque data *closure*.

`void` [Function]
`svz_coserver_ident_invoke` (*svz_socket_t* ***sock**, *svz_coserver_handle_result_t* **cb**, *void* ***closure**)

Enqueue a request for the ident coserver to resolve the client identity at *sock*, arranging for callback *cb* to be called with two args: the identity (string) and the opaque data *closure*.

To make use of coservers, you need to start the coserver interface by calling `svz_updn_all_coservers` once before, and once after, entering the main server loop.

`int` [Function]
`svz_updn_all_coservers` (*int* **direction**)

If *direction* is non-zero, init coserver internals. Otherwise, finalize them. Return 0 if successful.

If *direction* is positive, init also starts one instance each of the builtin servers. If negative, it doesn't.

5.2.10 Codec functions

The codec interface of the Serveez core API supplies routines for setting up socket structures to perform encoding or decoding of its receive or send buffers. It is a transparent layer of buffer transition. The interface itself tries to unify different types of codecs. In order to add a new codec the programmer needs to write some wrapper functions around the actual implementation to fulfill certain entry and exit semantics of this interface.

`int` [Function]
`svz_foreach_codec` (*svz_codec_do_t* ***func**, *void* ***closure**)

Call *func* for each codec, passing additionally the second arg *closure*. If *func* returns a negative value, return immediately with that value (breaking out of the loop), otherwise, return 0.

`svz_codec_t *` [Function]
`svz_codec_get` (*char* ***description**, *int* **type**)

Find an appropriate codec for the given *description* and *type* (one of either `SVZ_CODEC_ENCODER` or `SVZ_CODEC_DECODER`). Return `NULL` if there is no such codec registered.

`void` [Function]
`svz_codec_ratio` (*svz_codec_t* ***codec**, *svz_codec_data_t* ***data**)
Print a text representation of a codec's current ratio in percent if possible.

`int` [Function]
`svz_codec_register` (*svz_codec_t* ***codec**)
Register *codec*. Does not register invalid or duplicate codecs. Return zero on success, non-zero otherwise.

`int` [Function]
`svz_codec_unregister` (*svz_codec_t* ***codec**)
Remove *codec* from the list of known codecs. Return zero if the codec could be successfully removed, non-zero otherwise.

`int` [Function]
`svz_codec_sock_receive_setup` (*svz_socket_t* ***sock**, *svz_codec_t* ***codec**)
Arrange for *sock* to decode or encode its receive data via *codec*. Return zero on success, non-zero otherwise.

(You need to have set the `check_request` method previously for this to work.)

`int` [Function]
`svz_codec_sock_receive` (*svz_socket_t* ***sock**)
"This routine is the new `check_request` callback for reading codecs. It is applied in `svz_codec_sock_receive_setup`. Usually it gets called whenever there is data in the receive buffer. It lets the current receive buffer be the input of the codec. The output buffer of the codec gets the new receive buffer of *sock*. The old `check_request` callback of *sock* gets called afterwards. When leaving this function, the receive buffer gets restored again with the bytes snipped consumed by the codec itself." [ttn sez: huh?]

`int` [Function]
`svz_codec_sock_send_setup` (*svz_socket_t* ***sock**, *svz_codec_t* ***codec**)
Arrange for *sock* to encode or decode its send buffer via *codec*. Return zero on success, non-zero otherwise.

(You need to have properly set the `write_socket` member of *sock* previously for this to work.)

`int` [Function]
`svz_codec_sock_send` (*svz_socket_t* ***sock**)
"This is the new `write_socket` callback for *sock* which is called whenever there is data within the send buffer available and *sock* is scheduled for writing. It uses the current send buffer as input buffer for the codec. The output buffer of the codec is used to invoke the `write_socket` callback saved within `svz_codec_sock_send_setup`. After this the send buffer is restored again without the bytes consumed by the codec." [ttn sez: huh?]

`int` [Function]
`svz_codec_sock_disconnect` (*svz_socket_t* ***sock**)
Try to release the resources of both the receiving and sending codec of *sock*.

This callback is used as the `disconnected_socket` callback of the socket structure *sock*. It is called by default if the codec socket structure *sock* gets disconnected for some external reason.

`svz_codec_t *` [Function]
`svz_codec_sock_detect` (*svz_socket_t *sock*)
Return a valid codec detected by scanning the receive buffer of *sock*, or `NULL` if no codec could be detected.

5.2.11 Server types

As already noted in the main Serveez manual a server type is the main specification of the abilities and configuration items of a server which can be instantiated. It is represented by `svz_servertype_t` in Serveez. It contains server specific members like its name, different callbacks, a single default configuration and a list of configuration items which determine what can be configured.

5.2.11.1 Macros for setting up a new server type

When specifying a server type you also need to define configuration items for it. These items refer to addresses in the example configuration of the server type. These macros can be used to define such items.

`SVZ_REGISTER_INT` (*name, item, defaultable*) [Macro]
Register a simple integer. C-type: `int`. The given *name* specifies the symbolic name of the integer and *item* the integer itself (not its address). The *defaultable* argument can be either `SVZ_ITEM_DEFAULTABLE` or `SVZ_ITEM_NOTDEFAULTABLE`.

`SVZ_REGISTER_BOOL` (*name, item, defaultable*) [Macro]
Register a boolean value. C-type: `int`.

`SVZ_REGISTER_INTARRAY` (*name, item, defaultable*) [Macro]
Register an array of integers. C-type: `svz_array_t *`.

`SVZ_REGISTER_STR` (*name, item, defaultable*) [Macro]
Register a simple character string. C-type: `char *`.

`SVZ_REGISTER_STRARRAY` (*name, item, defaultable*) [Macro]
Register a string array. C-type: `svz_array_t *`.

`SVZ_REGISTER_HASH` (*name, item, defaultable*) [Macro]
Register a hash table associating strings with strings only. C-type: `svz_hash_t *`.

`SVZ_REGISTER_PORTCFG` (*name, item, defaultable*) [Macro]
Register a port configuration. C-type: `svz_portcfg_t *`.

`SVZ_REGISTER_END` () [Macro]
Indicate the end of the list of configuration items. It is the only mandatory item you need to specify in an example server type configuration.

`SVZ_CONFIG_DEFINE` (*description, config, prototypes*) [Macro]
Expand to a data structure that properly associates the example configuration *config* with the name *description* and its configuration items *prototypes*, for use within a server type definition.

5.2.11.2 General server type functionality

The following set of functions are used to manage the list of known server types in the Serveez core library. Serveez itself uses some of these functions to register its builtin server types.

`int` [Function]
`svz_foreach_servertype` (*svz_servertype_do_t* ***func**, *void* ***closure**)
Call *func* for each servertype, passing additionally the second arg *closure*. If *func* returns a negative value, return immediately with that value (breaking out of the loop), otherwise, return 0.

`void` [Function]
`svz_servertype_add` (*svz_servertype_t* ***server**)
Add the server type *server* to the currently registered servers.

`svz_servertype_t *` [Function]
`svz_servertype_get` (*char* ***name**, *int* **dynamic**)
Find a servertype definition by its short name. If *dynamic* is set to non-zero, try to load a shared library that provides that servertype. Return `NULL` if no server with the given variable prefix *name* has been found.

`svz_servertype_t *` [Function]
`svz_servertype_find` (*svz_server_t* ***server**)
Find a given server instances *server* server type. Return `NULL` if there is no such server type (which should never occur since a server is a child of a server type).

5.2.11.3 Dynamic server loading

The core API of Serveez is able to register server types dynamically at runtime. It uses the dynamic linker capabilities of the underlying operating system to load shared libraries (or DLLs on Win32). This has been successfully tested on Windows and GNU/Linux. Other systems are supported but yet untested. Please tell us if you notice misbehaviour of any sort.

`void` [Function]
`svz_dynload_path_set` (*svz_array_t* ***paths**)
Set the additional search paths for the serveez library. The given array of strings gets `svz_free`d.

`svz_array_t *` [Function]
`svz_dynload_path_get` (*void*)
Create an array of strings containing each an additional search path. The loadpath is hold in the environment variable '`SERVEEZ_LOAD_PATH`' which can be set from outside the library or modified using `svz_dynload_path_set`. The returned array needs to be destroyed after usage.

5.2.12 Server functions

A server in Serveez is an instantiated (configured) server type. It is merely a copy of a specific server type with a unique server name, and is represented by `svz_server_t` in the core library.

5.2.12.1 Server functionality

This section contains functions dealing with the list of known servers in the core library of Serveez, also with the basics like creation and destruction of such servers.

`void` [Function]
`svz_foreach_server` (*svz_server_do_t* ***func**, *void* ***closure**)
Call *func* for each server, passing additionally the second arg *closure*.

`svz_server_t *` [Function]
`svz_server_find` (*void* ***cfg**)
Find a server instance by the given configuration structure *cfg*. Return `NULL` if there is no such configuration in any server instance.

`svz_array_t *` [Function]
`svz_server_clients` (*svz_server_t* ***server**)
Return a list of clients (socket structures) which are associated with the given server instance *server*. If there is no such socket, return `NULL`. Caller should `svz_array_destroy` the returned array.

`svz_server_t *` [Function]
`svz_server_get` (*char* ***name**)
Get the server instance with the given instance name *name*. Return `NULL` if there is no such server yet.

`int` [Function]
`svz_updn_all_servers` (*int* **direction**)
If *direction* is non-zero, run the initializers of all servers, returning -1 if some server did not think it is a good idea to run. Otherwise, run the local finalizers for all server instances.

5.2.12.2 Configuration

These functions provide an interface for configuring a server. They are used to create and modify the default configuration of a server type in order to create a server configuration.

`int` [Function]
`svz_config_type_instantiate` (*char* ***type**, *char* ***name**, *char* ***instance**, *void* ***options**, *svz_config_accessor_t* ***accessor**, *size_t* **ebufsz**, *char* ***ebuf**)
Instantiate a configurable type. The *type* argument specifies the configurable type name, *name* the name of the type (in the domain of the configurable type) and *instance* the instance name of the type. Return zero on success, otherwise -1.

`void` [Function]
`svz_config_free` (*svz_config_prototype_t* ***prototype**, *void* ***cfg**)
Release the configuration *cfg* of the given configuration prototype *prototype*. If *cfg* is `NULL`, do nothing.

`void *` [Function]
`svz_collect` (*int* `type`, *size_t* `count`, *void* `*data`)
: Create a collection of *type*, given the *count* items of *data*. Valid values of *type* are one of: `SVZ_INTARRAY`, `SVZ_STRARRAY`, `SVZ_STRHASH`. For a string hash, *data* should be alternating keys and values; the returned hash table will have `count / 2` elements. The C type of *data* for an int array should be `int[]`, and for string array or hash it should be `char*[]`. On error (either bad *type* or odd *count* for string hash), return `NULL`.

Here are some convenience macros for `svz_collect`:

`SVZ_COLLECT_INTARRAY` (*cvar*) [Macro]
: Return an integer array `svz_array_t *` created from `int cvar[]`.

`SVZ_COLLECT_STRARRAY` (*cvar*) [Macro]
: Return a string array `svz_array_t *` created from `char *cvar[]`.

`SVZ_COLLECT_STRHASH` (*cvar*) [Macro]
: Return a string hash `svz_hash_t *` created from `char *cvar[]`.

5.2.12.3 Bindings

The following functionality represents the relationship between port configurations as described in Section 5.2.13 [Port config funcs], page 79, and server instances. When binding a server to a specific port configuration the core library creates listeners as needed by itself.

`int` [Function]
`svz_server_bind` (*svz_server_t* `*server`, *svz_portcfg_t* `*port`)
: Bind the server instance *server* to the port configuration *port* if possible. Return non-zero on errors, otherwise zero. It might occur that a single server is bound to more than one network port if, e.g., the TCP/IP address is specified by '`*`' (asterisk) since this gets expanded to the known list of interfaces.

`svz_array_t *` [Function]
`svz_server_portcfgs` (*svz_server_t* `*server`)
: Return an array of port configurations to which the server instance *server* is currently bound to, or `NULL` if there is no such binding. Caller should `svz_array_destroy` the returned array when done.

`svz_array_t *` [Function]
`svz_server_listeners` (*svz_server_t* `*server`)
: Return an array of listening socket structures to which the server instance *server* is currently bound to, or `NULL` if there is no such binding. Caller should `svz_array_destroy` the returned array when done.

`svz_array_t *` [Function]
`svz_sock_servers` (*svz_socket_t* `*sock`)
: Return the array of server instances bound to the listening *sock*, or `NULL` if there are no bindings. Caller should `svz_array_destroy` the returned array when done.

`int` [Function]
`svz_binding_contains_server` (*svz_socket_t* ***sock***, *svz_server_t* ***server***)
Checks whether the server instance *server* is bound to the server socket structure *sock*. Return one if so, otherwise zero.

`size_t` [Function]
`svz_pp_server_bindings` (*char* ***buf***, *size_t* ***size***, *svz_server_t* ***server***)
Format a space-separated list of current port configuration bindings for *server* into *buf*, which has *size* bytes. The string is guaranteed to be nul-terminated. Return the length (at most ***size*** `- 1`) of the formatted string.

5.2.12.4 Server core

`svz_t_handle svz_child_died` [Variable]
Set to a non-zero value whenever the server receives a SIGCHLD signal.

`int` [Function]
`svz_shutting_down_p` (*void*)
Return non-zero if the core is in the process of shutting down (typically as a result of a signal).

`int` [Function]
`svz_foreach_socket` (*svz_socket_do_t* ***func***, *void* ***closure***)
Call *func* for each socket, passing additionally the second arg *closure*. If *func* returns a negative value, return immediately with that value (breaking out of the loop), otherwise, return 0.

`svz_socket_t *` [Function]
`svz_sock_find` (*int* ***id***, *int* ***version***)
Return the socket structure for the socket id *id* and the version *version*, or `NULL` if no such socket exists. If *version* is -1 it is not checked.

`int` [Function]
`svz_sock_schedule_for_shutdown` (*svz_socket_t* ***sock***)
Mark socket *sock* as killed. That means that no further operations except disconnecting and freeing are allowed. All marked sockets will be deleted once the server loop is through.

`int` [Function]
`svz_sock_enqueue` (*svz_socket_t* ***sock***)
Enqueue the socket *sock* into the list of sockets handled by the server loop.

`void` [Function]
`svz_sock_setparent` (*svz_socket_t* ***child***, *svz_socket_t* ***parent***)
Set the *child* socket's parent to *parent*.

This should be called whenever a listener accepts a connection and creates a new child socket.

`svz_socket_t *` [Function]
`svz_sock_getparent` (*svz_socket_t* ***child***)
Return the *child* socket's parent socket structure, or `NULL` if this socket does not exist anymore. This might happen if a listener dies for some reason.

`void` [Function]
`svz_sock_setreferrer` (*svz_socket_t* ***sock***, *svz_socket_t* ***referrer***)
Set the referring socket structure of *sock* to *referrer*. If *referrer* is `NULL` the reference will be invalidated.

This can be used to create some relationship between two socket structures.

`svz_socket_t *` [Function]
`svz_sock_getreferrer` (*svz_socket_t* ***sock***)
Get the referrer of the socket structure *sock*. Return `NULL` if there is no such socket.

`svz_portcfg_t *` [Function]
`svz_sock_portcfg` (*svz_socket_t* ***sock***)
Return the parent's port configuration of *sock*, or `NULL` if the given socket has no parent, i.e. is a listener.

5.2.12.5 Server loop

This section describes the main server loop functionality. There two modes of operation. The default mode as used in Serveez is to jump into the loop and wait until the core library drops out of it. In the other mode, the caller tells the Serveez core library to scan (and process) its socket chain once and return immediately. Thus, caller is able to issue additional functionality in between each pass, useful if such functionality cannot be handled within the timers (notifiers) of servers and sockets.

`void` [Function]
`svz_loop_pre` (*void*)
Initialize top-of-cycle state.

Call this function once before using `svz_loop_one`.

`void` [Function]
`svz_loop_post` (*void*)
Clean up bottom-of-cycle state.

Call this function once after using `svz_loop_one`.

`void` [Function]
`svz_loop` (*void*)
Loop, serving. In other words, handle all signals, incoming and outgoing connections and listening server sockets.

`void` [Function]
`svz_loop_one` (*void*)
Handle all things once.

This function is called regularly by `svz_loop`.

5.2.12.6 Server sockets

This section deals with creating and handling listeners. These functions provide the default routines invoked when accepting a new connection on a listener. This is necessary for connection oriented protocols (TCP and named pipes) only.

[FIXME: All funcs internalized! Write something else here!]

5.2.13 Port configurations

A port configuration is a structure defining a network or file system configuration. Depending on the type of a server, it can be bound to one or more port configurations. There are two major types of port configurations: connection oriented (TCP and PIPE), and packet oriented (ICMP, UDP and RAW).

`struct sockaddr_in *` [Function]
`svz_portcfg_addr` (*svz_portcfg_t *port*)

Return the pointer of the `sockaddr_in` structure of the given port configuration *port* if it is a network port configuration. Otherwise return `NULL`.

`in_port_t` [Function]
`svz_portcfg_port` (*svz_portcfg_t *port*)

Return the UDP or TCP port of the given port configuration or zero if it neither TCP nor UDP.

`char *` [Function]
`svz_portcfg_ipaddr` (*svz_portcfg_t *port*)

Return the pointer to the ip address `ipaddr` of the given port configuration *port* if it is a network port configuration. Otherwise return `NULL`.

`char *` [Function]
`svz_portcfg_device` (*svz_portcfg_t *port*)

Return the network device name stored in the given port configuration *port* if it is a network port configuration. Return `NULL` if there is no such device set or if the port configuration is not a network port configuration.

Serveez maintains an internal list of port configurations, with each identified by its name. When you bind a server to a port configuration, it does not get bound to a certain name but to its content. If there are two or more port configuration specifying the same network or file system configuration just a single one gets actually used.

`svz_portcfg_t *` [Function]
`svz_portcfg_create` (*void*)

Create a new blank port configuration.

`int` [Function]
`svz_portcfg_equal` (*svz_portcfg_t *a*, *svz_portcfg_t *b*)

Check if two given port configurations structures are equal, i.e. specifying the same network port or pipe files. Return `SVZ_PORTCFG_EQUAL` if *a* and *b* are identical, `SVZ_PORTCFG_MATCH` if the network address of either port configuration contains the other (INADDR_ANY match), and otherwise `SVZ_PORTCFG_NOMATCH` or possibly `SVZ_PORTCFG_CONFLICT`.

`svz_portcfg_t *` [Function]
`svz_portcfg_add` (*char *name*, *svz_portcfg_t *port*)
Add the given port configuration *port* associated with the name *name* to the list of known port configurations. Return `NULL` on errors. If the return port configuration equals the given port configuration the given one has been successfully added.

`svz_portcfg_t *` [Function]
`svz_portcfg_get` (*char *name*)
Return the port configuration associated with the given name *name*. Return `NULL` on errors.

`void` [Function]
`svz_portcfg_destroy` (*svz_portcfg_t *port*)
Make the given port configuration *port* completely unusable, removing it from the list of known port configurations. Do nothing if *port* is `NULL`.

`int` [Function]
`svz_portcfg_mkaddr` (*svz_portcfg_t *this*)
Construct the `sockaddr_in` fields from the `ipaddr` field. Return zero if it worked. If it does not work, the `ipaddr` field did not consist of an ip address in dotted decimal form.

`svz_portcfg_t *` [Function]
`svz_portcfg_dup` (*svz_portcfg_t *port*)
Make a copy of the given port configuration *port*.

5.2.14 Boot functions

The most important functions are `svz_boot` and `svz_halt` which must be the first and the last call to the core API.

`void` [Function]
`svz_boot` (*char const *client*)
Initialize the core library. *client* is typically a program's `argv[0]`. If `NULL`, take it to be '`anonymous`'.

`long` [Function]
`svz_uptime` (*void*)
Return the number of seconds since `svz_boot` was called, or -1 if `svz_boot` has not yet been called.

`void` [Function]
`svz_halt` (*void*)
Finalization of the core library.

5.2.14.1 Runtime parameters

There are several runtime parameters indicating the abilities of the libserveez core API:

`SVZ_RUNPARM_VERBOSITY`
The log-level verbosity.

`SVZ_RUNPARM_MAX_SOCKETS`
: Maxium number of clients allowed to connect.

These are manipulated by `svz_runparm` and two convenience macros, both of which accept *nick*, a C token without the prefix '`SVZ_RUNPARM_`' (e.g., `VERBOSITY`).

`int` [Function]
`svz_runparm` (*int* `a`, *int* `b`)
: Set or get a runtime parameter. If *a* is -1, return the value of runtime parameter *b*. If *a* specifies a runtime parameter, set it to *b* and return 0. Otherwise, return -1.

`SVZ_RUNPARM` (*nick*) [Macro]
: Return the value of runtime parameter *nick*.

`SVZ_RUNPARM_X` (*nick*, *val*) [Macro]
: Set the runtime paramater *nick* to have value *val*, an integer.

5.2.15 Network interface functions

The network interface functions of the Serveez core API allow access to the network devices on your system. The system administrator can set up these devices to be bound to different Internet addresses and thereby split the network configuration into different *domains*. Thus, the system is able to separate the traffic of different networks. If set up correctly, Serveez can follow these rules.

`int` [Function]
`svz_foreach_interface` (*svz_interface_do_t* *`func`, *void* *`closure`)
: Call *func* for each interface, passing additionally the second arg *closure*. If *func* returns a negative value, return immediately with that value (breaking out of the loop), otherwise, return 0.

`int` [Function]
`svz_interface_add` (*size_t* `index`, *char* *`desc`, *int* `family`, *const void* *`bits`, *int* `detected`)
: Add a network interface to the current list of known interfaces. Drop duplicate entries. The given arguments *index* specifies the network interface index number, *desc* an interface desription, *family* an address-family (e.g., `AF_INET`), *bits* the address data in network-byte order, and the *detected* flag if the given network interface has been detected by Serveez itself or not.

5.2.16 Useful Windows functions

Serveez is meant to run on Windows systems as well (with some restrictions of course). These functions are available with the Windows implementation of the Serveez core API only. They allow access to the Windows registry database and some other useful things.

`int` [Function]
`svz_windoze_daemon_control` (*char* *`prog`)
: If *prog* is non-NULL, start the daemon thread with it. Otherwise (if *prog* is NULL), stop the daemon thread. Return 0 on success, -1 on failure.

`WCHAR *` [Function]
`svz_windoze_asc2uni` (*CHAR *asc*)
Convert an ASCII string into a UNICODE string.

`CHAR *` [Function]
`svz_windoze_uni2asc` (*WCHAR *unicode*)
Convert a UNICODE string into an ASCII string.

6 Porting issues

Serveez was always designed with an eye on maximum portability. Autoconf and Automake have done a great job at this. A lot of `#define`'s help to work around some of the different Unix' oddities. Have a look at `config.h` for a complete list of all these conditionals.

Most doubtful might be the Win32 port. There are two different ways of compiling Serveez on Win32: Cygwin and MinGW. The Cygwin version of Serveez depends on the Unix emulation layer DLL `cygwin1.dll`. Both versions work but it is preferable to use MinGW for performance reasons. The Cygwin version is slow and limited to a very low number (some 64) of open files/network connections.[1]

There are major differences between the Win32 and Unix implementations due to the completely different API those systems provide.

Processes and Threads
: Because process communication is usually done by a pair of unidirectional pipes we chose that method in order to implement the coservers in Unix. The Win32 implementation are threads which are still part of the main process.

Sockets and Handles
: On Win32 systems there is a difference in network sockets and file descriptors. Thus we had to implement quite a complex main socket loop.

Named Pipes
: Both systems Unix and Win32 do provide this functionality (Windows NT 4.0 and above). The main differences here are the completely different APIs. On a common Unix you create a named pipe within the filesystem via `mkfifo`. On Win32 you have to `CreateNamedPipe` which will create some special network device. A further difference is what you can do with these pipes. On Win32 systems this '`network device`' is valid on remote machines. Named pipes on Unix are unidirectional, on Win32 they are bidirectional and instantiatable.

Winsock Versions
: There are some difference between the original Winsock 1.1 API and the new version 2.2.x. In a nutshell, WinSock 2 is WinSock 1.1 on steroids, it's a superset of 1.1's APIs and architecture. In addition to its new features, it also clarifies existing ambiguities in the 1.1 WinSock specification and adds new extensions that take advantage of operating system features and enhance application performance and efficiency. Finally, WinSock 2 includes a number of new protocol-specific extensions. These extensions –such as multicast socket options– are relegated to a separate annex, since the main WinSock 2 protocol specification is protocol-independent.

  The Winsock DLL and import library for version 1.1 are `wsock32.dll` and `wsock32.lib` and for version 2.2 it is `ws2_32.dll` and `ws2_32.lib`. Serveez is currently using version 2.2.

  The Winsock API is still a bit buggy. Connected datagram behaviors are not pertinent to any WinSock 2 features, but to generic WinSock. On Win95

[1] This was written circa 2003—maybe the situation is now improved.

it is possible to use `recvfrom/WSARecvFrom` on a "connected" UDP socket, but on NT4 `recvfrom/WSARecvFrom` fail with 10056 (`WSAEISCONN`). NOTE: `sendto/WSASendTo` fail with `WSAEISCONN` on both (which I do not see any reason for, but anyway ...).

Raw sockets on Windows systems

Raw sockets require Winsock 2. To use them under Windows NT/2000, you must be logged in as an Administrator. On any other Microsoft's we were trying to use the `ICMP.DLL` (an idiotic and almost useless API) without success. Microsoft says they will replace it as soon as something better comes along. (Microsoft's been saying this since the Windows 95 days, however, yet this functionality still exists in Windows 2000.) It seems like you cannot send ICMP or even raw packets from the userspace of Windows (except via the `ICMP.DLL` which is limited to echo requests). We also noticed that you cannot receive any packets previously sent. The only thing which works on all Windows systems (9x/ME/NT/2000/XP) is receiving packets the "kernel" itself generated (like echo replies). One good thing we noticed about Windows 2000 is that the checksums of fragmented ICMP packets get correctly recalculated. That is not the case in the current Linux kernels.

Miscellaneous

To use the Win32 Winsock in the Cygwin port, you just need to `#define Win32_Winsock` and `#include "windows.h"` at the top of your source file(s). You will also want to add `-lwsock32` to the compiler's command line so you link against `libwsock32.a`.

What preprocessor macros do I need to know about ? We use `_WIN32` to signify access to the Win32 API and `__CYGWIN__` for access to the Cygwin environment provided by the dll. We chose `_WIN32` because this is what Microsoft defines in VC++ and we thought it would be a good idea for compatibility with VC++ code to follow their example. We use `_MFC_VER` to indicate code that should be compiled with VC++.

Why we do not use pipes for coservers ? Windows differentiates between sockets and file descriptors, that is why you can not `select` file descriptors. Please `close` the pipe's descriptors via `CloseHandle` and not `closesocket`, because this will fail.

The C run-time libraries have a preset limit for the number of files that can be open at any one time. The limit for applications that link with the single-thread static library (`LIBC.LIB`) is 64 file handles or 20 file streams. Applications that link with either the static or dynamic multithread library (`LIBCMT.LIB` or `MSVCRT.LIB` and `MSVCRT.DLL`), have a limit of 256 file handles or 40 file streams. Attempting to open more than the maximum number of file handles or file streams causes program failure.

As far as I know, one of the big limitations of Winsock is that the SOCKET type is *not* equivalent to file descriptor. It is however with BSD and POSIX sockets. That is one of the major reasons for using a separate data type, SOCKET, not an int, a typical type for a file descriptor. This implies that

you cannot mix SOCKETs and stdio, sorry. This is the case when you use `-mno-cygwin`.

Actually they are regular file handles, just like any other. There is a bug in all 9x/kernel32 libc/msv/crtdll interface implementations `GetFileType` returns `TYPE_UNKNOWN` for socket handles. Since this is AFAIK the only unknown type there is, you know you have a socket handle. There is a fix in the more recent perl distributions that you can use as a general solution. `-loldnames -lperlcrt -lmsvcrt` will get you `TYPE_CHAR` for socket handles.

Now follows the list on which operating systems and architectures Serveez has been build and tested successfully.

- FreeBSD 3.3, 4.0, 4.3, 4.4, 4.5, 4.6, 4.7 on Intel
- FreeBSD 4.0, 4.4, 4.6 on Alpha
- NetBSD 1.5, 1.6 on Alpha
- OpenBSD 3.0, 3.1 on Alpha
- OpenBSD 3.1 on Intel
- GNU/Linux 2.x.x on Intel
- GNU/Linux 2.x.x on Alpha
- GNU/Linux 2.2.x on Sparc64 (UltraSparcII + MachV)
- GNU/Linux 2.x.x on PowerPC (RS/6000)
- GNU/Linux 2.4.x on ia64 (Itanium)
- GNU/Linux 2.4.x on StrongARM (iPAQ)
- GNU/Linux 2.4.x on Motorola 680x0
- GNU/Linux 2.4.x on IBM S/390
- GNU/Linux 2.4.x on HP PA-RISC
- GNU/Linux 2.4.x on MIPS
- Solaris 2.6, 2.7 on Sparc32, Sparc64
- SunOS 5.8 on Sparc32
- SunOS 5.6 on Intel
- AIX 4.3 on RS/6000
- MacOS X10.1/Darwin 1.4, 5.4, 5.5 on PowerPC (Macintosh)
- Windows 95 on Intel
- Windows 98 on Intel
- Windows Millennium Edition on Intel
- Windows NT 4.0 on Intel
- Windows 2000 on Intel
- Windows XP on Intel
- IRIX 6.x on MIPS
- Tru64/OSF1 UNIX V4.0 (former Digital UNIX) on Alpha
- Tru64/OSF1 UNIX V5.x (former Digital UNIX) on Alpha
- HP-UX B.11.11 on HP PA-RISC (PA-8700/PA-8600)
- HP-UX B.11.22 on ia64 (Itanium)

7 Bibliography

This section contain some of the documents and resources we read and used to implement various parts of this package. They appear in no specific order.

1. RFC 760
   The Internet Protocol
2. RFC 1071
   Computing the Internet Checksum
3. RFC 1413
   Identification Protocol
4. RFC 1459
   Internet Relay Chat Protocol
5. RFC 1945
   Hypertext Transfer Protocol – HTTP/1.0
6. RFC 2068
   Hypertext Transfer Protocol – HTTP/1.1
7. RFC 2616
   Hypertext Transfer Protocol – HTTP/1.1
8. RFC 768
   User Datagram Protocol
9. RFC 791
   Internet Protocol
10. RFC 777
    Internet Control Message Protocol
11. `http://www.mingw.org/`
    Home of the MinGW (Minimal GNU for Windows) project
12. `http://gnutelladev.wego.com/`
    The Gnutella Protocol
13. `http://www.efnet.org/`
    The official EFNet site (includes Hybrid IRC server)
14. `http://www.sockets.com/`
    Winsock Development Information
15. `http://tangentsoft.net/wskfaq/`
    Winsock Programmer's FAQ

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

S

T